

AD-A093 788

MCDONNELL DOUGLAS ASTRONAUTICS CO HUNTINGTON BEACH CA

F/8 9/2

METRICS OF SOFTWARE QUALITY.(U)

NOV 80 Z JELINSKI, P MORANDA, J CHURCHWELL

F49620-77-C-0099

UNCLASSIFIED

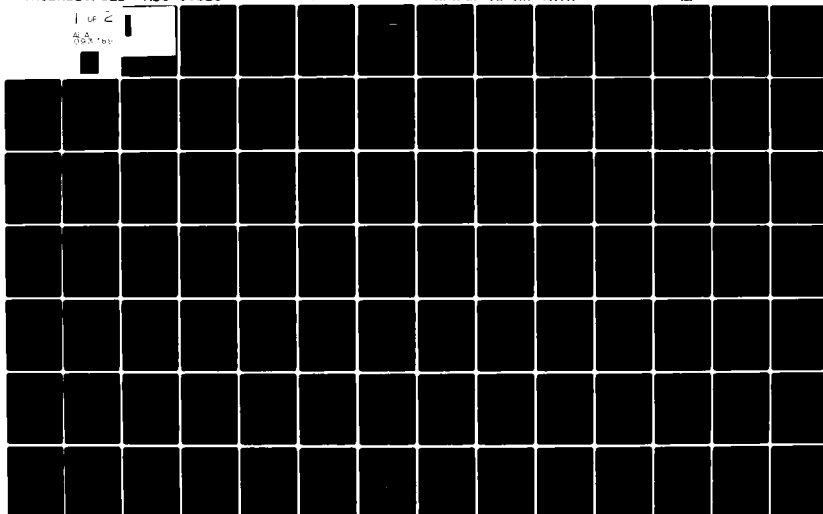
MOC-69326

AFNCR-TD-RD-1376

MI

1 of 2

64-100



AD A093788

302214-01 (08 JAN 70)

DOC FILE COPY

MCDONNELL DOUGLAS ASTRONAUTICS COMPANY

MCDONNELL DOUGLAS

CORPORATION

81

039

12



**METRICS OF SOFTWARE QUALITY  
FINAL TECHNICAL REPORT**

NOVEMBER 1980

MDC G9326

DTIC  
UNCLASSIFIED  
1991 5 12

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (AFSC)  
NOTICE OF TRANSMITTAL TO DDC  
This technical report has been reviewed and is  
approved for public release IAW AFR 190-12 (7b).  
Distribution is unlimited.  
A. D. BLOSE  
Technical Information Officer

**MCDONNELL DOUGLAS ASTRONAUTICS COMPANY-HUNTINGTON BEACH**  
5301 Bolsa Avenue Huntington Beach, California 92647 (714) 896-3311

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19 REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM	
1. REPORT NUMBER (18) AFOSR-TR-80-1376	2. GOVT ACCESSION NO. AD A093788	3. RECIPIENT'S CATALOG NUMBER (9) Final rept.	
4. AUTHOR(s) (6) METRICS OF SOFTWARE QUALITY.	5. TYPE OF REPORT & PERIOD COVERED 1 Jun 1977-30 Oct 1980 Final Report	6. PERFORMING ORG. REPORT NUMBER (14) MDC-G9326Y	
7. AUTHOR(s) (10) Zygmund Jelinski, P. B./Moranda, and J. B./Churchwell	8. CONTRACT OR GRANT NUMBER(s) (15) F49620-77-C-0099	9. PERFORMING ORGANIZATION NAME AND ADDRESS McDonnell Douglas Astronautics Company 5301 Bolsa Avenue Huntington Beach, CA 92647	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS (16) 611021-2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Scientific Research Bolling Air Force Base District of Columbia 20332	12. REPORT DATE November 1980	13. NUMBER OF PAGES 130	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) (12) 140	15. SECURITY CLASS. (of this report) Unclassified	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE (17) A2	
16. DISTRIBUTION STATEMENT (of this Report)  Approved for public release; distribution unlimited			
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)			
18. SUPPLEMENTARY NOTES			
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)  Software Metrics Test Tools			
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)  This report covers the period from 1 June 1977 to 30 October 1980. A major task on this contract was to make a comprehensive review of the literature on software metrics and of quantitative measures of  (Continued on next page)			

DD FORM 1473

EDITION OF 1 NOV 65 IS OBSOLETE

389310

iii

UNCLASSIFIED  
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

program testing.) The original review is contained in the first Interim Report (MDC G7517, dated July 1978); this review has been slightly revised and updated in this report.

Another accomplishment was the development of an automatic procedure for testing FORTRAN programs with random numbers and random symbols. This procedure first drives the program with sets of random data, then senses the tracks taken, compares each generated track against all predecessors, then on the basis of the pattern of occurrence of new tracks, provides an estimate of the total number of residual paths. Additional sets of random numbers can be then generated.

The program developed to instrument a given FORTRAN program and provide data for evaluating coverage is described in the Final Report of earlier work (MDC G6553, dated December 1976). Changes which have been made are described herein.

In the related topics of Software Reliability, two methods of estimating the residual error content of an entire program on the basis of data obtained in the testing of portions of it have been developed and are detailed here.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

F49620-77-C-0099

Accession For  
LIFE GRANT  
MAY 1960  
CIVIL RIGHTS  
BIRMINGHAM

✓

A

## ABSTRACT

A major task on this contract was to make a comprehensive review of the literature on software metrics and of quantitative measures of program testing. The original review is contained in the first Interim Report (MDC G7517, dated July 1978); this review has been slightly revised and updated in this report.

Another accomplishment was the development of an automatic procedure for testing FORTRAN programs with random numbers and random symbols. This procedure first drives the program with sets of random data, then senses the tracks taken, compares each generated track against all predecessors, then on the basis of the pattern of occurrence of new tracks, provides an estimate of the total number of residual paths. Additional sets of random numbers can be then generated.

The program developed to instrument a given FORTRAN program and provide data for evaluating coverage is described in the Final Report of earlier work (MDC G6533, dated December 1976). Changes which have been made are described herein.

In the related topics of Software Reliability, two methods of estimating the residual error content of an entire program on the basis of data obtained in the testing of portions of it have been developed and are detailed here.

## CONTENTS

Section 1	INTRODUCTION AND OVERVIEW	1
	1.1 Introduction	1
	1.2 Objectives and Tasks Descriptions	1
	1.3 Course of the Research Program	2
	1.4 Publications and Presentations	5
Section 2	LITERATURE REVIEW AND CRITIQUES OF SOFTWARE METRICS	7
	2.1 Reviews	7
	2.2 Critiques of Software Metrics	13
Section 3	COVERAGE BY RANDOM AND CONSTRUCTED CASES	33
	3.1 Introduction and Background	33
	3.2 Applications	56
	3.3 Additional Problems in Coverage Testing	80
Section 4	ERROR-DETECTION MODELS	87
	4.1 Summary	87
	4.2 Introduction	87
	4.3 Conclusions	95
	4.4 Glossary	95
	REFERENCES	97
Appendix A	AUGMENTED ORLA PROGRAM	A-1
Appendix B	APTS OUTPUT	B-1
Appendix C	OUTPUT FROM A CONSTRUCTED CASE AND CONSTRUCTED CASES LISTING	C-1

## FIGURES

1	Test Scores Program and Flow Diagram	36
2	Coding for Example Program	42
3	Combined Flow Chart and Code of Example Program	43
4	Augmented Code for Example Program	44
5	Flow Diagram of Miller and Spooner Example	47
6	Selected Path Through Example Program	48
7	Response to Zero Matrix	51
8	Response to Data Formed by Interchanging 1st and 2nd Rows of Original Matrices	52
9	Reversal Analysis	60
10	APTT Numbering for Program	61
11	APTS Segment Identification	62
12	Listing of an Example Program	76
13	Response to First Data	81
14	General Flow of Computation	82
15	Partially Pruned Flow Diagram	85
16	Purification Process and its Realization	89

## TABLES

I	ORLA Segment Usage Versus Trial Number	70
---	--	----

## Section 1 INTRODUCTION AND OVERVIEW

### 1.1 INTRODUCTION

The essential focus in this research has been the production of useful metrics to characterize the quality of software at any stage of its development. The end product desired is simply a set of metrics which have utility and a description of the means of applying them. The metrics selected are few in number. For completeness a list of all metrics which were found and considered as candidates at the time of the review in 1978 are included with comments on their prospective use.

The primary metric considered as an integral part of this research is coverage. Coverage can be described by a spectrum of choices: coverage at the branch level, at the instruction level, the segment level, "track" level, or execution path level. The procedures which are required to firmly establish the level of testedness or coverage of a software package form the substantive portion of this research.

In addition to coverage metrics, the software reliability models developed during an earlier contract and which have been employed to estimate mean times to failure and error content of a completed package, were modified so as to derive estimates on the basis of observations of errors during the initial phases of testing on incomplete or partial programs. Two models were developed, one is based on the assumption that a cumulative record is maintained of the percentage of the completed program that the (varying) tested portion represents, the other assumes that the count of the number of instructions under test is available.

### 1.2 OBJECTIVES AND TASKS DESCRIPTIONS

The original plan for the first phase of the research was given in terms of the tasks:

A. Review contemporary work of researchers in software testing field to postulate testing strategies.

B. Perform preliminary tests of selected programs to obtain some data on various testing strategies.

C. Evaluate parameters influencing software quality to suggest appropriate metrics.

D. Document as appropriate to facilitate later extensive experiments.

The primary effort during the second period encompassed work on three tasks:

A. Tailor or expand the testing programs that were developed in the first phase of the contract.

B. Code so as to provide valuations of the program predicates, and values of the artificial program variables which provide the data for the search procedures.

C. Modify, install, and test the tool on a laboratory computer when the scope and size of the test tool are established.

The subsequent work centered on the assessment of the practicality of a fully automated version of testing. The goal of this work was to test the tool and the methodology, using the several constructs (connection matrix, status vectors, predicate valuations, and input and output data) through the implementation on a "laboratory" type of computer, such as the Nanodata QM-1. The assessment of the practicality has been carried out but the implementation required programming effort which far exceeded budgeted labor.

### 1.3 COURSE OF THE RESEARCH PROGRAM

The initial effort was made on a literature review. The review made was very comprehensive and its scope is indicated by the list of papers, publications, and reports which were reviewed either in depth and critically, or in content. This list comprises Section 2.1. Subsequently, evaluations of the metrics which had potential ability were made. These are given in Section 2.2.

The substantive contribution to two aspects of software quality were initiated after the literature review. The metrics which showed promise, the two which could be most productively studied were the degree of coverage (or testedness) and MTTF (meantime to next error detection). These two metrics were exclusively studied in this research.

Previous successes in testing simple programs with random numbers led to the belief that such testing could serve as an efficient initial testing set on more complex programs. This testing has been found to be very much more effective than the sometimes trivial and always, limited in number, check problems, which are normally used (some of the cases generated by random test data for a polynomial-solving routine produced polynomials whose coefficients had ratios of  $10^{13}$  and this would be an essentially inconceivable choice for most testers). Accordingly, experiments were performed on progressively larger FORTRAN programs and the general tendency which was indicated on smaller programs was supported, random numbers generally produced good tests and in fact, in one of the later programs tested, the first 100 random number sets produced 99 different tracks.\*

A means of determining the number of residual tracks still existing has been developed in earlier work (Reference 1) but it required a time consuming matching or comparison procedure. One of the earliest programming tasks was to automate this procedure. Developed was an even further extension of the already augmented Program Evaluation and Tester (PET) tool, which itself was developed in 1974 (Reference 2). With the random number generator, which was an augmentation to PET made in earlier work (Reference 1), and distinguished by the name Program Testing System (PTS), and the automated comparison procedure, it was possible to produce sets of data in quantities approaching, to any desired degree, the asymptotic number of tracks which could ever be generated by random numbers. The modified program is designated as the Augmented Program Testing System (APTS).

---

\*A track derives its name from two facts: first, a segment of the associated execution sequence is counted only once even though it may have appeared with multiplicity in the execution sequence; second, the order of execution is immaterial and, for example, the sequences ABCAC and BACCA are equivalent.

The next step was to develop a systematic way of executing the still remaining tracks by formation of what are called constructed cases. The framework in this was a procedure developed by W. Miller and D. L. Spooner (Reference 3). This consists in converting the problem of unguided searching for data input points which will drive a particular path or track, into a systematic process related to a common optimization problem. Auxiliary variables are inserted at predicate sites, and a certain simple function of these variables is evaluated for each input data set, when a data set is found which produces a positive value in the function, the path associated with the preassigned predicate valuations then will have been exercised; and, if the trial data set does not produce a positive value, then any of several search techniques commonly employed in optimization problems can be used to determine subsequent trial data sets. The initial work in this application was all performed without computer assistance.

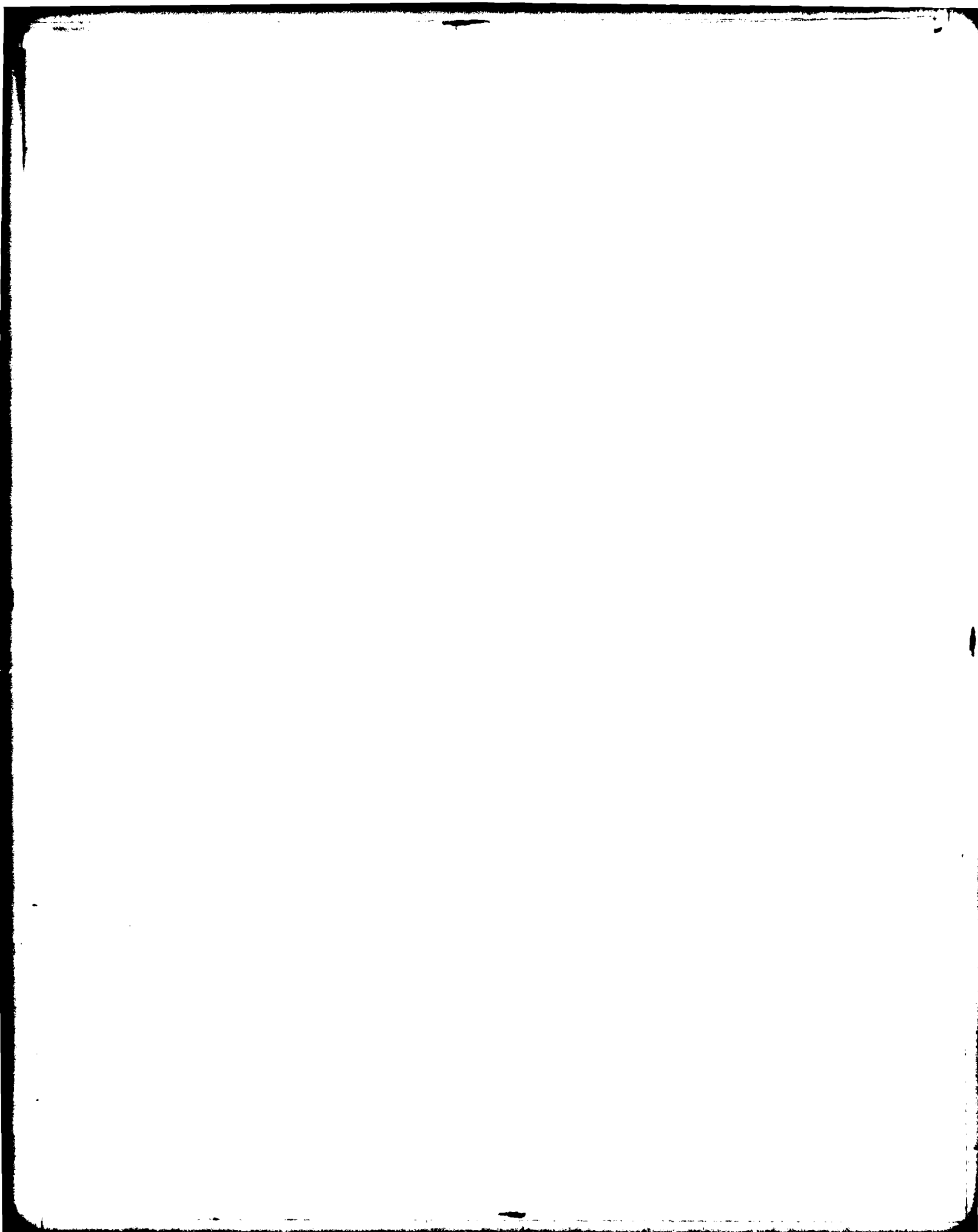
The final major goal was to incorporate both the random testing and constructed cases into one comprehensive sequential testing process. Starting with random numbers, these would be employed until new cases become difficult to find, at which point a transfer to testing by constructed cases would be made, using displays to show the predicate sites and the unexercised branches (valuations of the predicates). Auxiliary variables would then be inserted, the program would be recompiled, starting- or trial-data would be used, the composite objective function would be evaluated, and a search procedure invoked. The latter two steps would be carried out in an iterative fashion until the selected path was achieved or judged to be infeasible. Unfortunately, the magnitude of the programming effort required to implement the display/operator/computer complex was judged to be too extensive and expensive to carry out. Accordingly, only portions of the implementation have been developed. These are described in this report.

With respect to the study of the MTF, it was carried out in a low priority status, throughout the course of the research. Interfaces were made at several conferences with most of the analysts who have worked in the field of software error modelling. Two significant new models of the error-making process were developed during the third year of this study.

#### 1.4 PUBLICATIONS AND PRESENTATIONS

The following are the major publications or presentations of research sponsored in whole or in part by the Air Force Office of Scientific Research:

1. P. B. Moranda, "Limits to Program Testing with Random Number Inputs", Proceedings of COMSAC 1978, November 13-15, Chicago, Illinois.
2. P. B. Moranda, "Event-Altered Rate Models for General Reliability Analysis", IEEE Transactions on Reliability, Vol R-28, No. 5, December 1979.
3. Presentation by P. B. Moranda of a paper, "On the Modelling of the Error Process", to the 1st Minnowbrook Workshop on Software Performance Evaluation, sponsored by Syracuse University at Rome Air Development Center, October 1978.
4. P. B. Moranda, "Error Detection Models for Application During Program Development", Proceedings of Pathways of System Integrity, ACM Meeting Gaithersbury, MD, June 1980.
5. Presentation by P. B. Moranda of same paper at National Computer Conference, Anaheim, California, 22 May 1980.
6. Presentation by P. B. Moranda of same paper to 3rd Minnowbrook Workshop on Software Performance Evaluation, sponsored by Syracuse University and Rome Air Development Center, August 1980.
7. Presentation by Z. Jelinski of a paper "An Approach to Solution of Problems with Support Software as Deliverables" to Defense Systems Management Review, Ft. Belvoir, Virginia, March 1978.
8. P. B. Moranda, "Asymptotic Limits to Program Testing, INFOTECH State-of-the-Art Report on Program Testing, INFOTECH 1979.



Section 2  
LITERATURE REVIEW AND CRITIQUES OF SOFTWARE METRICS

2.1 REVIEWS

Two different levels of review were made, one is thorough and complete at an analytical level, the other for content only.

2.1.1 In Depth Reviews

2.1.1.1 Review of Work Published Prior to July 1978

Rather extensive and detailed examinations were made of the literature of the software testing field and of software metrics in general. The following papers were reviewed indepth during the period June 1977 to June 1978. Informal reviews of the following listed papers were provided to the contractor.

1. TRW Software Reliability Study. TRW Final Report, RADC, TR-76-236, August 1976.
2. M. L. Shooman, "Structural Models for Software Reliability Predictions", Proceedings of the 2nd International Conference on Software Engineering, 13-16 Oct 1976, San Francisco, California.
3. H. E. Williams, T. A. James, A. A. Beauregard, and P. Hilcoff, "Software Reliability Systems: A Raytheon Project History", RADC-TR-77-188, Final Technical Report, June 1977.
4. IBM Federal Systems Division, "Statistical Prediction of Programming Errors", RADC-TR-77-175 Final Technical Report, May 1977.
5. Doty Associates, Inc., "Software Cost Estimation: Vol 1", RADC-TR-77-220, Final Technical Report, June 1977.
6. J. R. Brown, H. N. Buchanan, "The Quantitative Measurement of Software Safety and Reliability" SDP 1776, 24 August 1973.
7. M. Shooman and A. Laemmel "Statistical Theory of Computer Programs - Information Content and Complexity" Digest of Papers Fall COMPCON 77, Washington, D. C., 6-9 September 1977.
8. G. J. Schick and R. W. Wolverton, "An Analysis of Competing Software Reliability Models" IEEEETSE, March 1978; Vol. SE-4, No. 2.

9. G. J. Myers, Software Reliability, Wiley-Interscience, 1976.
10. A. Fitzsimmons and T. Love, "A Review and Evaluation of Software Science", ACM Computing Surveys Vol. 10, No. 1, March 1978.
11. B. Littlewood and J. L. Verrall, "A Bayesian Reliability Growth Model for Computer Software", Record of 1973 Symposium on Computer Software Reliability, New York, N.Y., 1973.

#### 2.1.1.2 Additional Reviews

1. A. L. Goel and K. Okumoto, "Bayesian Software Prediction Models, Vol I: An Imperfect Debugging Model for Reliability and Other Quantitative Measures of Software Systems", RADC-TR-78-155, Rome Air Development Center, N.Y., 1978.
2. J. D. Musa, "Progress in Software Reliability Measurements", Proc. 2nd Software Life Cycle Management Workshop, Atlanta, Georgia, August 1978.
3. R. E. Schafer, et al, "Validation of Software Reliability Models", Hughes Aircraft Co., RADC-TR-79-147, June 1979.
4. W. D. Brooks, R. W. Motley, "Analysis of Discrete Software Reliability Models", IBM Corp., RADC-TR-80-84, RADC, New York, April 1980.
5. E. H. Forman and N. D. Singpurwalla, "An Empirical Stopping Rule for Debugging and Testing Computer Software", Journal of the American Statistical Association, Vol 72, December 1977.
6. A. N. Sukert, "An Investigation of Software Reliability Models", Proc. 1977 Annual Rel. Maint. Symp., Philadelphia, PA, 1977.

#### 2.1.2 Literature Reviewed for Content

1. Z. Manna. Mathematical Theory of Computation, McGraw-Hill, Inc., New York, 1974.
2. T. Gilb, Software Metrics, Winthrop Publishers, Inc., Cambridge, Mass., 1977.
3. A. Goel. "Bayesian Software Predictions Models," RADC-TR-77-112, March 1977.
4. M. Shooman, "Manpower Deployment Effects on Software Error Models," in RADC-TR-76-143, May 1976.
5. Boeing Computer Services, "Software Data Acquisition," RADC-TR-77-130, April 1977.
6. W. H. Howden, "Methodology for Generation of Program Test Data," IEEE TransComp, Vol. C-24, May 1975.
7. L. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," IEEE TSE, SE-2, 1976.

8. S. Gerhart and L. Yelowitz, "Fallibility in Applications of Modern Programming Techniques," IEEEETSE Vol. SE-2, No. 3, Sept. 1976.
9. R. F. Serfozo, "Compositions, Inverses, and Thinning of Random Measures," Syracuse University, Dept. of Ind. Eng. and Ops. Research, December 1975.
10. L. Osterweil, "Depth-First Search Techniques and Efficient Methods for Creating Test Paths," Univ. of Colorado Dept. of Comp Sci TR No. CU-CS-077-75, August 1975.
11. W. Miller and D. Spooner, "Automatic Generation of Floating Point Test Data," IEEEETSE Vol. SE-2, No. 3, Sept. 1976.
12. G.E.P. Box and K. B. Wilson, "Attainment of Optimum Conditions," J. Royal Stat Soc., Vol. XIII, No. 1, 1951.

### 2.1.3 Review of Testing Tools and Procedures

Articles of a review nature have identified and briefly described a large number of different testing tools. D. J. Reifer (Reference 4) identified 70 different types of tools and briefly discussed each type. C. V. Ramamoorthy and S. F. Ho (Reference 5) discuss, in some detail, 15 different tool types. A review of these different types here would be duplicative. Instead a composite review of the limited number of reports listed above dealing with the testing process will be presented. Usually the potential deficiencies of the processes or tools are brought out in the description, but not their advantages.

Before the discussion of individual classes of tools is undertaken here, it is well to note that the paper by Goodenough and Gerhart (Reference 6) illuminates many of the heretofore neglected points concerning testing.

Some of the important points they make in this respect are:

- A. It is not enough to execute a statement with a particular set of conditions, it must be tested in all combinations of conditions;
- B. In the same sense, a path through a loop may have to be taken several times before the conditions for error revelation are met;
- C. Missing, but required-for-correctness, components of a program (such as predicates or assignments) clearly cannot be identified by "cover-testing" a program;
- D. Generally a program must be examined for what it actually does instead of what the tester is told the program does and, at each point of

interface, it must be examined for what it can do;

E. The environment, including the operating system, hardware processor and language, have to be examined.

#### 2.1.3.1 Inside Out Testing

Several different techniques have been employed to develop test cases on the basis of a specified set of valuations or outcomes of the program's predicate. The mathematical expressions employed in the program predicates, are used to develop a set of restrictions on the input data space. Solution of the set of equations then produces a point or set of points that will achieve the path through the program. The difficulty with the procedure is that the set of equations involved often are not tractable, even for cases where only "area" (as distinct from point) solutions are required. This difficulty is, to a degree, alleviated by use of interactive entry of data and display-guided solutions.

#### 2.1.3.2 Symbolic Execution

Instead of operating on numerical (or logical) values for variables in a processing, the program's operations can be carried out on the symbols themselves. This technique was independently proposed by W. E. Howden (Reference 7) at McDonnell Douglas Astronautics Company, B. Elspas, et al. (Reference 8) at Stanford Research Institute, J. C. King of IBM (Reference 9) and Lori A. Clark (Reference 10) of the University of Massachusetts.

Programs, so exercised must be augmented so they become capable of symbolic execution of expressions and provide means for selecting specified branches or paths in them. Howden employs a system (DISSECT) processing the program that is to be symbolically executed, along with a list of commands that cause symbolic execution.

The advantages of symbolic execution are clear. In certain cases the printout consists of an explicit formula that is unambiguous to the reader. If the formula is correct, the program is correct for all data and there is no necessity for numerical comparisons or independent checks.

In many cases, however, the output is far from clear to any but the most experienced users. There is, for example, sometimes a need to maintain the

list of possible antecedents (a suspense file) for a program variable having several different symbols and values assigned to it. Further there is a context-dependency that a given assignment may have, caused, for example, by different encounters of an assignment during looping. This must be accounted for, and in the case of DISSECT, the context is identified by a number representing the dynamic instruction number (as distinct from the static sequence number associated with a listing). These and more complex problems have been faced by Howden and others and they provide finished products that are proof that such techniques can be used to good effect when the tools are in the hands of experts.

While there are some barriers to the "field" use of such techniques, they do not seem insurmountable and it is probably reasonable to expect that symbolic execution can be of common use.

#### 2.1.3.3 Automated Verification Systems

Several systems instrumenting a given program to permit the tallying of the uses of its instructions, branches, and so forth, are classified as automated verification systems by Reifer (Reference 4). They are usually not automated in the strict sense of the word: although they require a set of input test data to drive the program, there is no instantaneous feedback to change the data to test new unexercised sections of the program. A complaint on word usage can be also made that these systems do not really verify the tested program, and generally do not even consider the output in respect to its accuracy, or even its relevance.

A McDonnell Douglas Automation Company tool, called PET (for Program Evaluator and Tester) described by L. G. Stucki in a company report (Reference 2) and in the open literature (Reference 11), is typical of this class.

For a given data set, PET reports the usage by instruction and branch, which the execution sequence represents. There are other useful metrics, including the range of value for each of the program variables. Lists of unexercised program components also are printed out.

An augmented version of PET, that formed segments consisting of

"dynamically contiguous" program instructions, was used and described in a recent AFOSR-sponsored study (Reference 1). In that study, as with most other applications of PET, the emphasis is on the "coverage" of the tested program. Repeated tests with randomly generated input data were used, and their effects merged to produce a composite (montage) of the testing status of the program. Unexercised segments were used to find the governing predicate or predicates in the program listing, and so-called constructed cases were then formed. The process was continued as far as deemed possible to establish the testing degree.

This class of program monitors is useful in another way. Frequently exercised portions of a program can be identified by the tallies or counts and the identified regions can be examined to see if improvements can be made in the coding or basic algorithms.

#### 2.1.3.4 Automatic Test Generators

Conceivably any particular segment of a program has some input data that will cause it to be exercised. Since it is possible, as indicated in the section on inside-out testing, to back up from a particular point in the program to the "top," it should be possible to choose a set of inputs that will cause any given segment to be exercised. The technique used amounts to an identification of the program variables that are "active" at the segment, and then to relate these to the input variables. This is illustrated in Section 3.1.2 where the precise set of relations to the input data are developed explicitly from a particular "straight line" path through the program.

Usually it will not be necessary to develop the precise relations (which, it is noted, is almost the same as symbolic execution) between the program variables and the input, and it is only necessary to identify those inputs affecting the selected program variable. This can be accomplished in an even less elegant way by simply generating random numbers to serve as values for the input variables.

Whatever scheme is used, the automatic test generators provide a basis for economically meaningful testing-to-"completion." The idea is simply to

form a "feedback" loop between a cumulative record of the segments previously tested to, what might be called, a scenario generator. The scenario generator would provide a one-at-a-time selection for the untested segments, and the standard test generators could be used to "find" the required data. This will then cause the new scenario update and a new selection. This idea is mentioned again later in connection with the use of "tracks" and the automatic case generation process.

#### 2.1.3.5 Domain-Testing Strategies

The point mentioned above, in connection with the possible creation of a truly automatic test tool, brings up the important problem identified earlier, the essential impossibility of producing a particular numerical value by the usual kinds of random number generation. This is not a problem in estimating the asymptotic limits to testing with such numbers, because of the infrequent occurrence of these numbers in the sample. For the development of constructed cases where exhaustive testing can be achieved, it is necessary to specify the set of points in the input space which, after processing, will produce a specified value for a program variable.

Generally speaking, the particular set of points achieving the specified value has relatively small dimensionality (a point in two-space, a line in three-space, etc.) making the problem of testing boundaries important.

E. I. Cohen and L. J. White of Ohio State University (Reference 12), have investigated this and similar problems and developed strategies that will test domains with linear and non-linear boundaries (the latter only in two dimensions at present) in efficient ways. As noted, work of this kind is essential to any ultimately automatic testing scheme.

## 2.2 CRITIQUES OF SOFTWARE METRICS

### 2.2.1 Software Science Metrics

In the review paper by A. Fitzsimmons and T. Love (Reference 13) the principal metrics employed in Software Science are discussed in some detail. They are few in number: length, volume, program level, language level, effort and time.

A troubling feature of these metrics is that they are all based on counts of operators and operands and, as noted in Reference 14, there are many cases where it is not at all clear what particular mix of these fundamental elements a given program instruction represents. The effects of this lack of precision in the definition of operator and operand has been studied by J. L. Elshoff (Reference 15). In this study all of the primary metrics are computed for some 34 different programs for each of eight different interpretations of the way in which the counts of programming elements should be taken. These eight different methods produced exceptional variability in the metrics in cases where there was a significant effect in the vocabulary definitions. For example, program No. 13 which is the largest program, showed counts of 185 and 746 for operations and operands, respectively, under the first interpretation, and counts of 118 and 900 for the second interpretation. The effects on the metrics under the two interpretations are:

estimated length	9,645	versus	8,512 (11.7% smaller)
volume	91,902		82,373 (10.3% smaller)
level	0.00365		0.00212 (41.7% smaller)
minimum volume	334.6		174.2 (47.9% smaller)
effort	25.243		38.945 (54.2% larger)
global level	1.1037		0.607 (45% smaller)

The variation in these metrics is indicative of the effect that the subjective choices (8 different types) can cause. In a separate comparison, the single metric, effort, for the 8 options (for program No. 1) were: 0.783, 0.881, 0.937, 1.010, 1.065, 0.764, 0.794, 0.679. This variability, which is over 50% (from min to max) is evidence of a lack of "objectivity" in this (and other) measures.

#### 2.2.1.1 Complexity (Software Science Interpretation)

In the abstract of the paper by Fitzsimmons and Love it is noted that complexity of programs can be measured by the theory of Software Science. It was difficult to locate precisely where in the paper this complexity is measured because the word appears only incidentally in the text. It was determined, by direct inquiry, that it was measured by the effort metric.

This use of an extensive measure for complexity is indeed novel and does not correspond to intuition or to any other measures advanced by others. Halstead states that complexity of a program is measured by the total number of elementary discriminations required to produce it, and this count depends on the bulk of the program more than on its logical structure.

The previously published measures of complexity had to do with intensive measures such as the (normalized-to-unity) spectrum of the program listing across its indenture levels, or the density of branching statements.

The recently described measure of complexity by T. J. McCabe (Reference 16) is the cyclomatic number obtained from the flow graph of the program. This metric is described in a later section when the topic of complexity is re-examined. Suffice it to say, it is more an intensive measure than an extensive measure, and as McCabe points out (op.cit.) it is easy to write a program that is physically small but ultra complex.

The complexity measure of software science is directly related to the length of the program (the total number of operators and operands) and is finally developed on an absolute basis by the use of the so-called Stroud number, which is taken by M. Halstead to be 18 mental discriminations per second.

This Stroud number has, as its basis, some physical measurements of a human's ability to discriminate the frames of a kaleidoscopically presented visual sequence of images (related to the "flicker rate" in motion pictures). The use of this visual discrimination rate, as equal in value to the mental discriminations rate, is surely questionable.

#### 2.2.2 Software Metrics

Probably the best starting point for this discussion is a review of the metrics presented in the book by Tom Gilb (Reference 17). This serves more to cover the field than to make precise the concepts and definitions of the many metrics identified. Following this is a list of metrics having a reasonable likelihood of surviving through test and time.

### 2.2.2.1 Review of Gilb Metrics

#### Maintainability

The first definition offered by Gilb is that of maintainability. He defines it as

"the probability that, when maintenance action is initiated under stated conditions, a failed system will be restored to the condition within a specified time."

That definition is essentially the same as that used for hardware. In the hardware case the measure is almost always applied in a bottoms-up way, that is the maintainability is derived for each major assemblage from the records of its contained minor assemblies; the system's figure is derived from the major assemblies. Work records on the times to fix are estimated during design, and, once hardware is delivered, records are kept of the actual fix times.

Software should be amenable to the same broad guidelines. Some modules are likely to be more easily fixed than others and a better systems-wise figure can be developed from the bottoms-up composition. The records of maintenance of individual modules should be used to extrapolate for new errors. The fact that the process of error-finding tends to have long periods between finds (probably) does not alter the fundamental measure of the average time to fix. This is (probably) so because the late occurring errors are (probably) not of a different level of difficulty than the early occurring errors. (Should there be a trend towards longer fix times with the "age" of the error, a model would need to be developed).

#### Logical Complexity

Gilb's introduction into this topic identifies early work by L. Farr and H. J. Zagorski, who used the IF statement density as a measure of the logical complexity. Gilb also mentions "psychological" (his quotes) complexity of source programs and refers to some statistical work by L. M. Weissman which correlated metrizable program aids (comments, indentations, etc.) to productivity and accuracy.

### Structuredness

One of the metrics identified by Gilb is structuredness. This was one of many metrics proposed by TRW in a study for the National Bureau of Standards.

Structuredness is one of 12 low-level metrics identified by Gilb, the others are: device independence, completeness, accuracy, consistency, device efficiency, accessibility, communicativeness, self-descriptiveness, conciseness, legibility, and augmentability.

For structuredness, there are 9 submetrics which are, in actuality, questions concerning the existence of module size limits, program flow, and so forth. Gilb's Figure 51 (page 103) can be referred to for identification of the particular questions. It does not appear that the underlying metrics have any quantitative basis, and necessarily have either a zero or an all value.

Typical of a question, under a column headed "Definition of Metrics to Measure Structuredness," is: "Do all subprograms and functions have only one entry point?" Here, should the answer be no, there is no way of differentiating between "all-but-one" and "none."

Presumably a yes answer to all questions would indicate a perfectly structured program. Using these the characterizing features (from Figure 51 of Gilb) the program would be one which:

- A. Has rules for transfer of control between modules.
- B. Has limited modules sizes (Note: the limit is not specified).
- C. Has the ordering: commentary header block, specification statements, executable code (Note: it is hard to imagine a program that does not follow the order).
- D. Subprograms all contain, at most, one point of exit.
- E. Subprograms and functions all have only one entry point.
- F. Program flow is always forward, except where commented.
- G. Overlay structure is consistent with the subprogram's sequencing.
- H. Is subdivided into modules in accordance with readily recognized functions.
- I. Is written in standard constructs.

These submetrics are then scored as to their "correlation" with a "high score for the metric." The use of "correlation" as a descriptor for subjective judgment is highly questionable: there are no numbers to associate with the identified metrics, and the numbers associated with the "score," if present at all, are certainly vague.

Nonetheless, the "quantifiability" of the metrics is judged against six categories which, while neither exhaustive nor mutually exclusive, are nonetheless indicated as such by the tabular entries.

The other 12 metrics are probably treated in the same way as the structuredness metric, and, beyond their identification, do not appear to merit additional inquiry. (Self-descriptiveness, communicativeness, and accessibility, for example, appear to be invented to exercise the invention process, and do not represent useful metrics; others, such as augmentability, may have some value).

#### Reliability

Gilb's definition of system reliability is in close accord with the customary (hardware) definition. It states that "reliability is the probability that the system will perform satisfactorily (with no malfunctions) for at least a given time interval, when used under started conditions." This is modified only slightly under the definition offered later. Gilb's variation of his definition for system reliability when applied to program or software reliability are minor, a particular machine is denoted, and operations are "within design limits."

#### Repairability

The concept of repairability is a variation of the maintainability concept. The emphasis is on the probability of a repair within a specified time, when maintenance is performed under specified conditions. The requisite tools, parts, and men, are assumed to be available at the start, and this is one of the specified conditions.

### Serviceability

This metric is taken from hardware reliability and is the degree of ease or difficulty with which a system can be repaired. It is not considered quantifiable at present.

### Availability

Again, from the hardware reliability definition, this is the probability a system is operating satisfactorily at any point in time. It is usually measured by a ratio of times or mean times, and Gilb offers three variations of the concept (intrinsic-, operational-, and use-availability).

### Attack Probability

This metric is one of several Gilb suggests in the security aspects of programs. This metric is the probability of an attack (of a particular type) on a system during a particular time interval.

These attacks can be considered to be active (malicious) or passive (typified by invalid data).

### Security Probability

This is described by its alternative title, attack repulsion probability, and is a metric gauging the probability of a successful rejection in the system at any time. The attack type is specified. Gilb states that this concept is close to the concept of error detection probability. This is less true of active attacks (which may not persist) than it is for passive attacks such as bad data.

### Integrity Probability

This probability is the probability of no successful attack on the system:

$$I_g = 1 - [A_t \cdot (1 - S)]$$

where  $A_t$  is the attack probability for a particular time interval, and  $S$  is the probability of rejection (for all times).

### Accuracy

Several examples of the metric and a discussion contrasting it with precision are given by Gilb. The measurement ratio, correct data/all data, appears to be too vague for use involving, as it does, the idea of "correct data." Usually accuracy involves a continuum of values so that "correct" data is too narrowly defined for practical usage.

### Precision

The suggested measure of this metric, which aims to gauge the degree "to which errors tend to have the same root cause," is the ratio formed by dividing the number of actual errors at source, by the number of corresponding root errors observed in total caused by source bugs.

The difficulties in first knowing how many errors there are at the source seem unsurmountable, and tying together the "corresponding" errors with the source would not seem to be an easy task.

### Error Detection Probability

Gilb suggests a categorization of the error types and an assignment of the likelihood of detection of errors of the pre-specified type. The failure to include time aspects into the problem makes for a flawed definition. The probability of an eventual detection of an error is (probably) unity for almost all error types.

### Error Correction Probability

As defined by Gilb, this is the probability of reconstructing "data in the form and content originally intended." This is a vague concept when identification of the random event is sought. The originally intended form and content is generally not known, rather it develops as effects are judged unsatisfactory and tentative changes are made. There is a chance that the repair made will have an error that may lie undiscovered for a period of time, and so time should be involved in the measure in some way.

### Logical Complexity

In the text two metrics for logical complexity are identified, the number of binary decisions and the ratio of absolute logical complexity to

total complexity. But Gilb also suggests under the Figure 83 on page 161 that it be measured by the number of possible logical path combinations in a program.

In this respect Gilb illustrates with an unanswered question, the defect in using even the density of branching statements as a measure of complexity. In his Figure 84, two programs are shown, one which has 6 binary decision points and the other only one. But for a sufficiently large number of total instructions (say 239 as indicated in the description) the density of the clearly more complex program is less than the ultra-simple one. This alert is examined in the discussion of complexity later in this report.

#### Flexibility

Gilb defines this as that part of complexity that is useful, and it is the ratio of useful to total that is the metric.

#### Built-in Flexibility

This is defined as the ability of a program to immediately handle different logical situations. It must be built-in in order to respond without loss of time.

#### Adaptability (open-ended flexibility)

Gilb acknowledges the difficulty of originating a metric for this concept and suggests, as a tentative measure, the count of the linkages between modules. This is the same as the metric used later for structural complexity.

#### Tolerance

This is defined as the ability of the system to accept different forms of the same information as valid. The proposed metric is the count of the number of different variations that can be handled by the system, where variation means the different media, different formats of input, or logical variations (such as misspellings and synonyms).

### Generality

The "degree of applicability of a system within a stated environment" constitutes generality. Its measurement is subjectively assigned (0 to 1).

### Portability

This is defined as the ease of conversion of a system from one environment to another. The metric is obtained by first forming the ratio of the resources required to move the program to a target environment to the resources needed to create the program for the target environment, and then subtracting the ratio from unity. The result is the ratio of the cost difference to the creation cost and, on the extremes, agrees with an economic measure of portability, because for a zero-cost move the portability is unity, and for a cost equal to the creation cost, the portability value is zero.

### Compatibility

This attribute is, according to Gilb, related to the concept of portability, the difference being that portability is a characteristic of a single system whereas compatibility applies to an average over a class of systems. This distinction provides the metric, an average portability over the collection of program systems.

### Redundancy Ratio

This is the first of what are called structural metrics by Gilb. This ratio generally is formed by taking the actual count of quantities to the minimum possible count.

### Hierarchy

This structural metric describes the number of indenture levels and the spectrum of program elements across these levels.

### Structural Complexity

As noted earlier in the section concerning adaptability, this is measured by the number of modules (absolute) or the ratio of linkages to the total number of modules. This is an easy metric to derive for some languages

as Gilb shows. For FORTRAN the modules are counted by the number of subroutines and functions, and the number of linkages is the total of subroutine parameters and the references to the common area.

#### Modularity

Although modularity is stated to be a synonym for structural complexity, it seems to stress the number of modules and not the linkages.

#### Distinctness

Distinctness as defined by Gilb involves errors and, in fact, is measured by a ratio between the number of bugs in the module and the number that are common to the module and another ("simultaneously"). It is hard to see how this ties to the intuitive concept of uniqueness, particularly how errors are necessary components of distinctness.

#### Effectiveness

Among the performance metrics, effectiveness is listed first. It is a probability of "success" within a given time and specified environment. The "success" means meeting an operational demand. Gilb composed efficiency from three probabilities: reliability, readiness probability, and design adequacy (on a scale from 0 to 1).

#### Efficiency

This attribute is defined as the ratio of useful work to the total expended.

#### Cost

Among financial metrics are costs and its major subdivisions, fixed and variable. Gilb uses the terms capital and operational.

#### Time

Computer and "Human" time resource metrics.

#### Space Metrics

This is more commonly called the size of a program. It can be measured on an atomic level by bits and bytes and, on the more common scale, by the number of instructions.

### Information

Gilb says that information content of a program is not directly measurable, and suggests use of "useful data" as an indirect means for measurement.

### Evolution

This is a measure of the incremental change to a system during a time interval,  $t$ . If the change is so pervasive that it constitutes a substitution, the metric would have a value of unit.

### Stability

Stability is the complement of Evolution and it denotes the percentage of unchanged content of a program (over a specified time period).

#### 2.2.3 Candidate Metrics

Clearly some of the questions that should have been asked of the community several years ago are:

- A. Are any attributes worth study?
- B. Which attributes are useful?
- C. Can these be measured in a form useful to the community?

It is clear from inspection of the Gilb metrics that there are many that will not survive the tests required of practical gauges. Most of the 13 low-level metrics identified by Gilb have little hope of common usage. The discussion concerning structuredness, in that section, indicates that the concept is initially vague and becomes amorphous after its component parts are identified (in the form of questions).

Of the metrics listed above, the following are considered of primary value: reliability, complexity, cost and time.

Regarded as secondary in importance are: maintainability and availability.

Supplementing these metrics are some that history may judge to be of more value than any of the metrics identified above: mean-time-to-next error, mean-time-to-perfection, error content, testedness, and purification level.

Excepting the complexity metric, it does not seem necessary to amplify on the previous Gilb definitions, and the following subsections deal with the augmenting metrics.

#### 2.2.3.1 Mean-Time-To-Next-Error

Of primary importance in the testing of programs is the decision on whether or not to release a given module or program. A good guide to this choice lies in the time pattern of the errors found, whether this pattern lies in a data base metered by CPU units, hours, days, or weeks is not relevant (except as its potential future uses may have to be considered). If the time pattern indicates a steady state or constant error rate, or, even worse, shows an increasing failure rate, there is clearly no reason for releasing the module and much evidence to the contrary. Once a pattern of decreasing counts (per unit time) is achieved, any of several models can be applied to the data that the error pattern represents, and estimates of the mean-time-to-next-error can be obtained.

It is the magnitude of this mean-time-to-next-error, or more commonly called the mean-time-to-failure, MTTF (which for a certain probability distribution, and steady state conditions, is the same as the mean-time-between-failures [MTBF]), considered in the context of its expected use, that is important. For real-time systems, governing, for example, weapons or aircraft, the MTTF should be several times as large as the mission duration. The proper figure for the MTTF is determined by the reliability specified by the customer for the system.

Values for the MTTF are available in any of several models: Jelinski-Moranda, Shooman, Schick-Wolverton, Moranda Geometric Purification, Moranda Hybrid Geometric-Poisson.

Littlewood and Verrall (Reference 18) avoid MTTF and insist instead on percentiles (such as the median) of the distribution describing the time between errors.

It is important to note, that for all models the MTTF is a parameter that is changed by either event or time. The Jelinski-Moranda model has an MTTF, indexed by the dummy variable  $i$ , which increases at the occurrence of each error, and can be given in terms of the model parameters  $N$  and  $\phi$  as

$$MTTF_{J-M} = \frac{1}{[N-(i-1)]\phi}$$

The Schick-Wolverton model has an "instantaneous" MTTF depending on both time and event, and it has an averaged MTTF that is obtained from the first moment of the Rayleigh distribution for the time of next error. Thus,

$$MTTF_{S-W} = \sqrt{\frac{\pi}{2}} \left[ \frac{1}{(N-n)\phi} \right]^{1/2}$$

For the Geometric Purification model, the MTTF is

$$MTTF_{G-P} = \frac{1}{Dk^n}$$

where  $D$  is the failure rate for the first error,  $k$  is the geometric ratio which is used to obtain the error rates, and  $n$  is the number of found errors.

The Shooman MTTF is given by

$$MTTF_S = [C E_T - E_C(t)]^{-1}$$

where  $C$  is a proportionality constant,  $E_T$  is the total error content, and  $E_C(t)$  is the total number of errors found.

#### 2.2.3.2 Mean-Time-To-Perfection

Some models permit an estimate of the mean time required to achieve an error-free program. Generally this estimate is accompanied by a variance (standard deviation) that is so large that it has little or marginal utility. It is nonetheless a guide to management and it is changed, and generally made more precise, as more errors are discovered.

The simplest way to form this estimate is to sum the estimated MTTF's for all remaining errors; hence (using MTTP for mean-time-to-perfection), the estimate so formed for the Jelinski-Moranda model is:

$$MTTP_{J-M} = \frac{1}{\phi} \sum_{j=n}^{N-1} \frac{1}{N-j}$$

For the Schick-Wolverton Model,

$$MTTP_{S-W} = \sum_{j=n}^{N-1} \sqrt{\frac{\pi}{2}} \left[ \frac{1}{(N-j)\phi} \right]^{1/2}$$

This last formula is incorrectly given in the latest Schick-Wolverton paper (Reference 19).

The Shooman model does not permit an estimate of the MTTP because the failure rate of that model is a continuous exponential. The mean time to achieve a zero with the exponentially decreasing failure rate is infinite.

The Moranda Geometric Purification model also does not have a finite average time to perfection. Even though discrete, the failure rate does not attain a zero value.

The Littlewood and Verrall model, based on Bayesian adjustment, does not involve a parameter that can be directly related to the MTTF, and it is required that some alternative be found. This can be provided by any of the percentiles of the distribution formed by convolution of the family of related exponential distributions they use in their examples. It is necessary, however, to rely on the most recently available, "a posteriori" distribution for one of the two parameters, and to continue the assumption concerning the way that the sequence of values for the other parameters are related. It presents a difficult problem analytically and probably has practical objections.

The recent publication by A. L. Goel and K. Okumoto (Reference 20) has relevance to this and some of the other problems. Their work in the present context employs a family of distributions that are the same as those used by Jelinski and Moranda but with an essential difference, they assume an imperfect repair and account for it with a parameter,  $p$ , that is the same for all errors. Using these variations, the distributions of the "first passage" times (zero errors) and of the times to achieve various levels of purification are derived.

#### 2.2.3.3 Error Content

Three models can be used to derive estimates of the error count. The Jelinski-Moranda model accomplishes it through use of equations developed from the assumption that there is a direct proportion between error content and failure rate. The corresponding Shick and Wolverton assumption is that the failure rate is proportional to both the number of errors and the "debugging time." The Shooman model can be used at two or more separated time intervals to estimate the error content. From observations of the average MTTF for these intervals, parameters of the linear relation between failure rate and error content can be found by simultaneous equations (for two intervals) or by least squares (for three or more).

#### 2.2.3.4 Purification Level

Although some models do not measure error content and may not achieve a perfect state, there is a measure that, in some cases, can be used to describe the state of perfection achieved at a given point in time. For error-content models, the ratio of the number found to the total number (estimated) provides the reasonable estimate. For the Moranda Geometric Purification process, the purification state can be estimated by taking the ratio of the initial failure rate to the achieved failure rate.

The purification level or percentage is clearly of more value than the error content since the absolute number is, by itself, generally a poor indicator of status because it is size-of-program related.

The several estimators of the purification percentage are (in terms of their defined parameters):

Jelinski-Moranda	$\frac{n}{N} \times 100$
Schick-Wolverton	$\frac{n}{N} \times 100$
Shooman	$\frac{E_C}{E_T} \times 100$
Geometric Purification	$(1-k^n) (100)$

#### 2.2.3.5 Testedness (Degree to Which a Program Has Been Tested)

A metric of a different kind is represented by the degree to which the program has been tested. There are several different types of "coverage" for a program, where "coverage" means that the program "elements" have been executed.

E. C. Miller (Reference 21) presented a useful list of several different coverage types in a sequence reflecting the increasingly larger size of the covering unit. The lowest level of coverage is obtained when every statement is executed at least once, the next level is achieved when each segment associated with the explicit or implicit predicate outcome is executed. For complex programs involving nested loops, the test coverage may necessarily be limited to the exercising of the program so as to test, one time, all so-called boundaries and interiors of loops, it being assumed that all segments are exercised. (Boundaries are the entries and exits from a loop.) A higher order of coverage consists of multiple passes through loops, these are tests that iterate all loops up to a certain specified limit (even 1), and provides additional coverage. The ultimate test coverage (with several other types in between) exercises all logical paths through a program.

One additional type is afforded by the testing of the type described in the earlier work (Reference 1), where tracks were identified. Coverage by tracks is intermediate between segment coverage and logical path coverage.

The metric for any of the types is simply the ratio of the number of "elements" (defined as instructions, segments, branches, predicates, tracks or logical paths) to the total number of these elements.

A new concept was introduced by Moranda (Reference 1) where the difficulty of enumerating the number of different elements is avoided. By using random numbers, it is possible (under some assumptions that are reasonable or acceptable to some and questionable or unacceptable to others) to estimate the total number of program elements that will eventually be achieved by random numbers. This technique was used in the original work (Reference 1) to estimate the total number of "tracks," but could as easily be used to derive the asymptotic limit to the number of logical paths.

#### 2.2.3.6 Complexity

As noted in the discussion on the effort metric employed for complexity measures by the Software Science advocates, the use of an extensive measure for complexity runs counter to intuition. The total number of "elementary discriminations" required to produce a program does not seem to properly reflect the structural aspects of complexity, for a "straight-line" program (no loops) of extreme length would have a high effort value, but might be judged rather simple.

Other measures were suggested in that same discussion. The density of branching statements was suggested, but as noted by McCabe in Reference 16, the density, as measured by instruction count, may be misleading. In that reference, a reasonably complex program containing six branches had so many (hypothesized) instructions that the density of branching statements was less than a short straight-line program (with one branch).

It is clearly necessary to alter the concept, and base the metric on the segment counts, rather than the instruction counts. This is a reasonable

position to take because some segments may contain a very large number of instructions. As far as the intricacies or complexities of a program are concerned, all segments are the same and do not depend on the number they are comprised of.

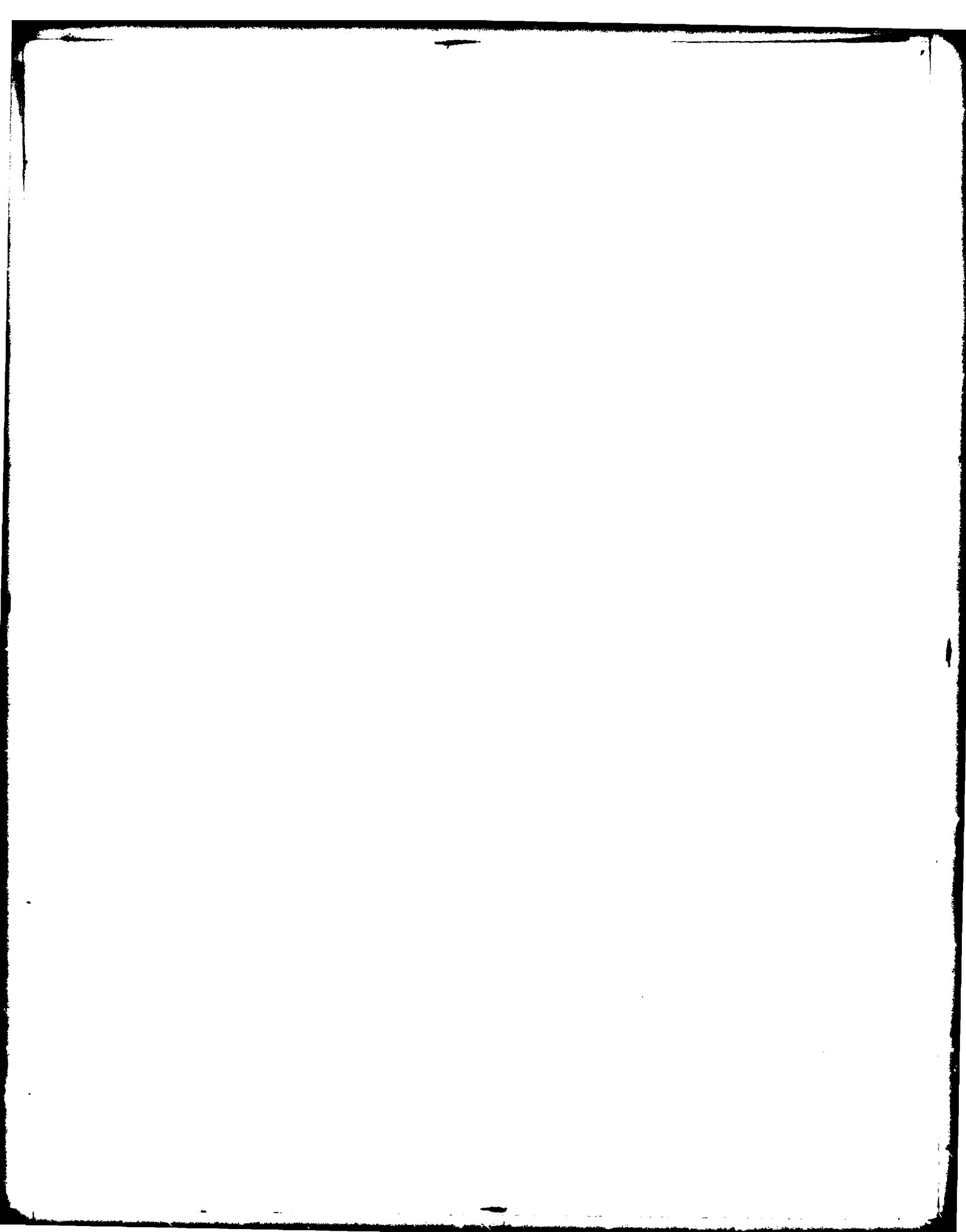
Thus, a more satisfactory metric for complexity would be either the number of segments or the number of logical paths. Since the latter are difficult to count in many cases, the former can be used, even though the way they connect is not measured thereby.

Another measure of complexity which may be of use is the indenture level spectrum. This concept is rather simple in that it tallies into each indenture level, each instruction of the program. By dividing the number in each category by the total number of instructions, a normalized-to-unity spectrum can be produced. There are clearly some deficiencies in this approach since a program that "shifts" back and forth between two adjacent levels is not judged to be more complex than one that has the same number of instructions at each level and "shifts" but once. The metric would require a complementary measure to provide a total measure of complexity.

A far better metric for complexity has been developed by T. J. McCabe (Reference 16). He suggests that the program be represented by a directed graph,  $G$ , in the usual way. The way the nodes (or vertices) and segments (edges) of  $G$  are connected is measured by a cyclomatic number, denoted by  $V(G)$ , determined by the number of edges, vertices, and connected components (where the latter is a subgraph of  $G$ ).

M McCabe proves a theorem that permits an alternative way of finding the cyclomatic number: for strongly connected graphs, the cyclomatic number is the maximum number of linearly independent circuits. In order to apply this theorem, it is necessary to form a strongly connected graph by looping back from the exit node to the entrance node.

It is generally easy to identify the cyclomatic number of most reasonably well-structured programs of small to moderate size. Where the program is extensive, the algebra set up by McCabe can be used to calculate the number.



### Section 3 COVERAGE BY RANDOM AND CONSTRUCTED CASES

#### 3.1 INTRODUCTION AND BACKGROUND

The following introductory section is essentially a duplicate of the text material used to introduce the topic of coverage in Reference 1.

##### 3.1.1 Framework for Representation

In the customary renditions of program flowcharts, each (rectangular) block represents either a simple instruction, or a group of operations, with a single output, while each diamond represents a single, explicit or implied, predicate which has two or more exit options. Connecting the blocks and diamonds of a flowchart, are directed lines denoted, and referred to, as arrows. These lines represent the options possible and are called flow-of-control arrows. These fundamental building blocks are adequate for the static or structural description of a program, but are not convenient for representing its operational aspects. The basic operations are better defined in terms of some simple program components. These lend themselves to mathematical descriptions and they motivate the choice for the "atomic" or fundamental unit of description.

First, it is noted that an instruction in a program, while easy to define (statically) in "machine language", becomes rather difficult in most of the higher order languages. Thus a "clear and add" instruction, in machine language, causes a register (accumulator) to be set to zero and another register to be transferred to the cleared register and nothing more. Once the final bit is transferred, the machine waits until the next instruction, which is generally started by a timing or clock pulse. On the other hand, the concept of an instruction in the higher languages is less clear. An "instruction" in ALGOL, for example, is either a statement or a declaration, and in either case is used to indicate required compiler (as against computer) actions. As a result of compiler action, an object program with computer interpretable instructions, is produced.

Thus, there is a spectrum of statements in that language: the simplest type is an assignment, such as  $X:=1$ ; while one of the more complex statements is, *begin ... end*, which groups statements together to form compound statements (and blocks).

In any higher order language where grouping is required, there is a need for so-called delimiters (explicit or implicit) which can be used as boundaries for the steps, and form the building blocks of a program. A similar device is required in the description of dynamic operations - a means of grouping instructions into fundamental operational units.

Generally, the linking of instructions can be represented by means of a Boolean indication, with the value 1 used where the instructions are or can be "contiguous", and 0 used to denote the fact that they are not connected. These Boolean values could be used as entries of a connection matrix whose row and columns are numbered to accord with an (arbitrary) numbering scheme for the steps. But a straightforward application in this manner, on the instruction level, would normally produce inordinately large and unmanageable connection matrices. Some of the redundant information in such a matrix could be eliminated if certain agreements can be made: for example, if step 1 is always followed in sequence by steps 2, 3, and 4 and there is not opportunity for branching until step 4 (at least), then steps 1 through 4 can be merged or combined, and three of the rows and columns of the connector matrix could be eliminated. This reduction in redundancy is an additional reason for choosing groups of instructions for the description.

Because certain instructions or statements have more than one output (such as *if...then...else*) there is a need to devise a convention which will permit identification of each of the exits. If statement A is a single-output statement and it connects to statement B which has multiple outputs, the notation  $[A,B)$ , which is "closed" on the left and "open" on the right, is meant to imply that A is executed and control is passed to (or toward) B, but that B is not executed, but it is next in line. If B is a two-output instruction and connects to  $L_1$  and  $L_2$ , then both  $[B,L_1)$  and  $[B,L_2)$  are used to describe the optional branches which can be taken.

The procedure which has been described can be illustrated by a flow diagram of a very simple program. In Figure 1 is a combination of a code listing on the right and a flow diagram on the left. Numbers refer to the instructions listed. The program is designed to process a sequence (one or more) of lists, with each list consisting of "test scores" augmented by the number -1 (which is not a test score); the last list is further augmented with a -2 (for HALT purposes). The program tallies the number of scores within each list which are at least as large as 70 (passing), and also tallies the total number of passing scores within all lists (the Grand Sum).

To continue with the description, it will be seen in Figure 1 that the first connection to a branching instruction is made at instruction number 3. From 3 the branch taken is determined by the predicate ( $X=-2$ ) and how the input to 3 (carried out of 2) values it (true or false). Thus, instruction number 3 is connected to 14 and to 4, as potential (operating) successors to 3. In the same way, 5 as a branching statement connects to 6 and 10.

A variation of the technique which is usually employed, characterized by connecting "nodes" (representing sets of instructions) is proposed here. Emphasis in this variation is on the branches which emanate or terminate with branching instruction, and, in fact, the fundamental or "atomic" element in the representation of a program is taken to be a segment or string of instructions between two branching instructions. More precisely a segment is: a sequence of instructions starting with either a START, or a branching instruction, and ending (but not inclusively) with the first subsequent branching instruction, or a HALT, in which particular case the segment is considered to include the instruction which ends it.

As an example of the way segments are developed, the flow diagram in Figure 1 is analyzed:

- $S_1 = [1,2,3)$
- $S_2 = [3,14,15)$
- $S_3 = [3,4,5)$
- $S_4 = [5,10,11,12,13,3)$
- $S_5 = [5,6)$
- $S_6 = [6,8,9,5)$
- $S_7 = [6,7,8,9,5)$

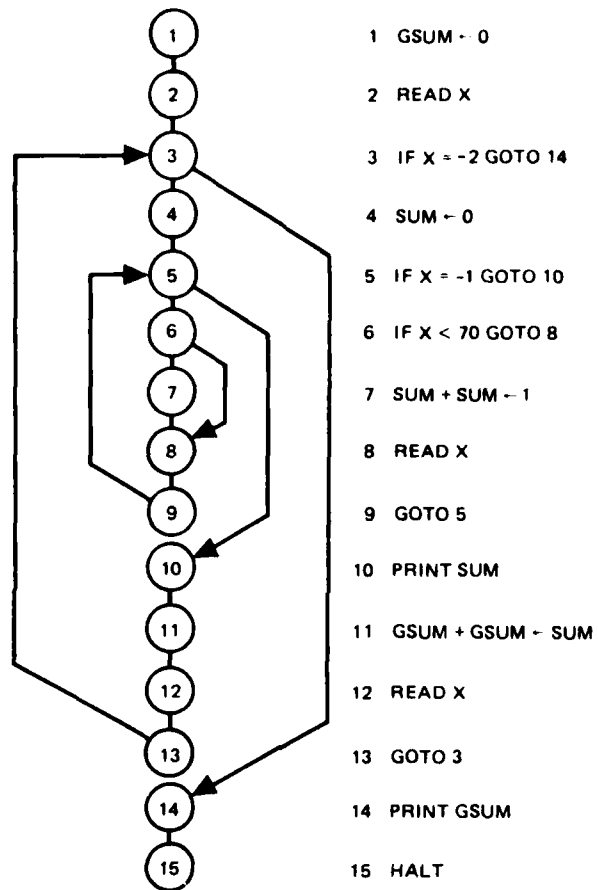


Figure 1. Test Scores Program and Flow Diagram

The distinction between brackets and parentheses is important and has been noted. The only cases where square brackets are used on the right are those in which the last instruction listed is a HALT (number 15 in the example).

Any particular set of values (for the coordinates) of the input vector (point in the input space), causes exactly one sequence of operations to be executed. These segments linked together form a logical path through the program. This is also called an execution sequence.

It is useful to modify the term logical path with the work realizable when input data can cause it. Before data is entered, possible (or feasible) logical paths can be formed by any concatenation of contiguous segments which have the START-segment first and end with a HALT-segment. In the case a program has self-contiguous segments (loops) or one or more concatenations which join end-to-end, the number of (possible) repetitions of the joined segments is arbitrarily large - except where a predetermined number of traversals are specified in the program.

The following sequences of segments in the program of Figure 1 are illustrative of some possible or feasible logical paths:

$S_1 S_2$   
 $S_1 S_3 S_4 S_2$   
 $S_1 S_3 S_4 S_3 S_4 S_3 S_5 S_6 S_4 S_2$   
 $S_1 S_3 S_5 S_6 S_4 S_2$   
 $S_1 S_3 S_5 S_7 S_4 S_2$

The first path is of minimum possible length, linking, as it does, the START - and HALT - segments. The last two are interesting in that they exhaust the collection of segments (but not the logical paths).

In order to determine realizable logical paths, the documentation or "program writeup" must be considered. In this simple case it is very easy to establish data which will realize the flows represented by the last two sequences of the above list. (It should be noted that insofar as testing to the instruction-level only one of these two need be driven but to obtain segment or branch-level testing, both need to be tested).

If for example the data sequence (stacked)

$x = 35, -1, -2$

is employed, the next to the last sequence of the above list describes the flow, and for the "stack"

$x = 75, -1, -2$

the last sequence describes the flow. The two stacks together provide an exhaustive test of the segments of the program.

Moreover, a single sequence 35, -1, 75, -1, -2 would also produce an exhaustive test of the segments with the sequence  $S_1 S_3 S_5 S_6 S_5 S_7 S_4 S_2$ . While these do not exhaustively test the realizable logical paths (which, without further explicit restrictions, are infinite in number), it is well to note that the complete segment-testing partially accomplishes one of the major purposes of case selection, that of exercising all instructions so as to uncover incompatibilities with the machine and other errors.

This limited form of testing brings up a very interesting and very obvious observation that is true for any program represented as a collection of segments: if a program consists of  $k$  segments, and every segment can be exercised by some data point, then only  $k$  data points are required to exhaustively test the program in the segment testing sense. This is of course very useful in the case that an interactive or communicative mode of testing is employed.

### 3.1.2 Extension to Testing for Track Coverage

Under AFOSR contract AF 44620-74-C-0008, MDAC developed a model which employs random numbers as input data, and, on the basis of the trial numbers on which "new" logical paths are driven by the input data, estimates the asymptotic, or eventual, level of testing achieved with random numbers. The basic analysis mechanism is the original Jelinski-Moranda model (Reference 22). The measurement used in the model is the number of trials

occurring between the discovery of new logical paths (rather than times between errors which comprised the raw data for the estimation of residual error content in the original application of the model).

There is another relevant use of this same model. If the probability law governing the selection of input data is known, then the coupling of information derived from sampling with universal (a priori) error rate data will permit an estimate of the operational reliability of the program. This procedure, also developed under the same AFOSR contract was reported in Reference 23.

A second model employing program or software input data for analysis, is due to TRW (Reference 24). In essence, this model uses a subdivision of the input data space into equivalence classes, each characterized by the particular logical path exercised by all of its members.

This subdivision was suggested earlier by W. Howden (Reference 7) and also by B. Elspas, et al., (Reference 8). In applications the TRW model has been used in the estimation of software reliability. The estimate is derived by composing the assumed-to-be-known probability that each subdivision is employed, with a sample-derived conditional probability of committing an error when the subdivision is used. The problem in the application of such a model is the difficulty involved in the formation of the subdivisions, confirmed by almost everyone who has attempted to work from a specified logical path to the descriptor of the input data associated with it. Another deficiency occurs when the model is used for estimation because, a permanent program is assumed which does not change to remedy the found errors. The problem of precisely carving out the equivalent classes is a severe barrier to application of such techniques. It is probably better to avoid the problem, as done in the application of random numbers described in Reference 1, or by using techniques like those described by W. Miller and D. Spooner (Reference 3).

The use of random numbers as inputs to a software package has fundamental limitations. For example the occurrence of an input which takes on a zero value is essentially impossible and this input, and others of a similar nature, must be supplied to produce a set of inputs which will achieve such values.

Nevertheless, as shown in earlier work (Reference 1), the fundamental limitation can be numerically estimated for a given program on the basis of the set of logical paths effected as a result of random drivers. It can be said that the number found in this way is an always fair and often an excellent bound on the total number of logical paths which are ever actually exercised.

The work of Miller and Spooner avoid these problems with an elegant substitute: instead of attempting to solve, in the input data space, the set of equations (or inequalities) associated with a specified logical path, they insert a new set of variables, one at each branching point in the program. An objective function of these variables is chosen so that when its functional value is positive, the input data is in the equivalence set associated with the specified logical path. This method employs standard procedures from the field of system optimization, starting with a randomly chosen initial point in the input domain.

For additional background, a review is made of the means of representing the flow graph by a connection matrix. As noted in prior work the matrix is constructed by assigning a 1 or 0 as an entry, according to whether or not there is a connection between the nodes (or segments) corresponding to the associated row and column of the matrix. A simple way of visualizing the problem of exhaustive testing can be posed in matrix format. Since a connection matrix  $C$  is a descriptor of potential links between segments, the execution sequence in response to an input data value  $x_1$  (in most applications  $x_1$  is a vector instead of a scalar), can be associated\* with a submatrix of  $C$ , say  $S_1$ . Since  $C$  is finite, the problem of exhaustive testing can be framed as follows: for  $C$  a given connection matrix find a set  $x_1, x_2, \dots, x_m$ , so that for the associated submatrices  $S_1, S_2, \dots, S_m$

$$C = \bigcup_{i=1}^m S_i$$

\*As discussed in Reference 1, an execution sequence can be mapped to submatrix by ignoring the ordering of its branches. This is valid only because of the definition used here for exhaustive testing.

where

$$\bigcup_{i=1}^m S_i$$

represents the Boolean union or sum of the  $S_i$ . (This essentially defines the nature of exhaustive testing at the track level).

An efficient test would be one in which the number of test points,  $m$ , is minimal.

As noted above, essentials of the process involve associating with each decision point (two-way) or predicate within the program, a function which has a non-negative value when the predicate is true, and negative value when it is false. In many cases, such as comparison between program variables by inequalities, the expression in the predicate can serve directly to define the function. If, for example, there is a test  $P \leq Q$ , then the variable assignment, or function,  $C = P - Q$ , can be used. Since the functions are relations among variables, they can be considered to be program variables. By forming variables of this kind at each branch point, the program is augmented in such a way that, in response to an input data set, an execution sequence will take place in which values are given not only to all program variables but also to the augmenting variables, which, as noted, are program variables.

Because the signs (+ or -) of the augmenting variables, set up a unique pattern for any input data, they can be used to define the equivalence classes mentioned above. It is noted again that in the formation of the equivalence classes the ordering of the sequence has been ignored.

In a different mode of usage, the sign of each of the augmenting variables can be specified in advance, and a point (or region) in the input data space causing this pre-specified pattern of signs can be sought. By assignment of any of a number of simple objective functions of the augmenting variables, with properties described subsequently, the problem can be stated as a search problem generally identified with optimization problems.

Generally the search is made to find data which will make the objective function positive; it is not necessary to achieve a maximum for the objective function, only that the value of the function be positive. This problem is much simpler than the optimization problem.

The technique due to W. Miller and D. Spooner (Reference 3) is illustrated by their example shown below. Their description of the example has been augmented in several ways.

The problem of the example is one of triangularization of an  $N \times N$  matrix by Gaussian elimination.

The original code is shown in Figure 2. A combined flowchart and code with predicates and branches identified, is shown in Figure 3. The predicates, shown enclosed in rectangular boxes are attached to the node representing the site of their occurrence. The augmented code employing the functions associated with the predicates, is shown in Figure 4. The input data to the program consists of the nine matrix entries:  $A(1,1), A(2,1), \dots, A(3,3)$ .

```

IP(N) = 1
DO 6 K = 1,N
  IF (K.EQ.N) GO TO 5
  KP1 = K+1
  M = K
  DO 1 I = KP1,N
    IF (ABS(A(I,K)).GT.ABS(A(M,K))) M = I
1  CONTINUE
  IP(K) = M
  IF (M.NE.K) IP(N) = -IP(N)
  T = A(M,K)
  A(M,K) = A(K,K)
  A(K,K) = T
  IF (T.EQ.O.) GO TO 5
  DO 2 I = KP1,N
2    A(I,K) = -A(I,K)/T
  DO 4 J = KP1,N
    T = A(M,J)
    A(M,J) = A(K,J)
    A(K,J) = T
    IF (T.EQ.O.) GO TO 4
    DO 3 I = KP1,N
3      A(I,J) = A(I,J) + A(I,K)*T
  4 CONTINUE
  5 IF (A(K,K).EQ.O.) IP(N) = 0
  6 CONTINUE
  RETURN
  END

```

Figure 2. Coding for Example Program

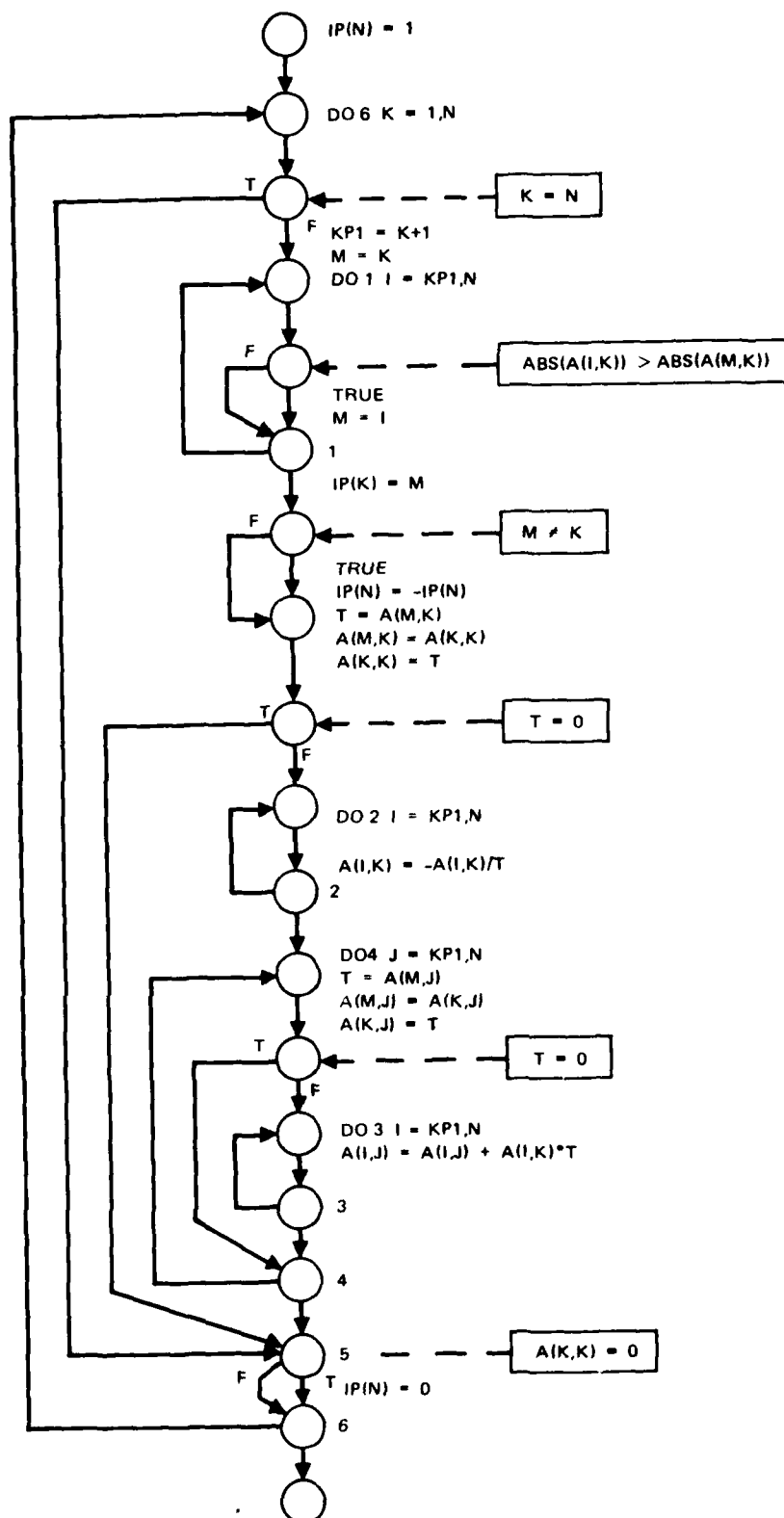


Figure 3. Combined Flow Chart and Code of Example Program

Formation of the augmented code is accomplished by making a straight line pass through the program under the assumption that the predicates inside the DO loop 1 are all true, and all of the rest are valued false. It is noted that the test results denoted as  $K=N$ , are governed by the input assignment to the matrix order,  $N$ , here taken in the example as  $N=3$ . There is a "false" valuation until  $K=3$ . These valuations are implicitly made in the construction of the program into a straight line representation. Similarly the tests, denoted as  $M=K$ , are completely determined by the tests in the DO loop 1 and do not explicitly show in the augmented code; they are used to develop the straight line code.

The variable  $C_1$ , shown on the first line of Figure 4, is positive if the predicate,  $ABS(A(2,1)) - ABS(A(1,1))$ , is true; and this condition has been specified as holding, since the predicate is in the DO loop 1. A similar remark applies to  $C_2$  and  $C_7$  in the straight line listing because they are repeats of the same test encountered under new conditions. On the other

8CR34

```

c1 = ABS(A(2,1)) - ABS(A(1,1)) > 0
c2 = ABS(A(3,1)) - ABS(A(2,1)) > 0
      T = A(3,1)
      A(3,1) = A(1,1)
      A(1,1) = T
c3 = ABS(T) > 0
      A(2,1) = -A(2,1)/T
      A(3,1) = -A(3,1)/T
      T = A(3,2)
      A(3,2) = A(1,2)
      A(1,2) = T
c4 = ABS(T) > 0
      A(2,2) = A(2,2) + A(2,1)*T
      A(3,2) = A(3,2) + A(3,1)*T
      T = A(3,3)
      A(3,3) = A(1,3)
      A(1,3) = T
c5 = ABS(T) > 0
      A(2,3) = A(2,3) + A(2,1)*T
      A(3,3) = A(3,3) + A(3,1)*T
c6 = ABS(A(1,1)) > 0
c7 = ABS(A(3,2)) - ABS(A(2,2)) > 0
      T = A(3,2)
      A(3,2) = A(2,2)
      A(2,2) = T
c8 = ABS(T) > 0
      A(3,2) = -A(3,2)/T
      T = A(3,3)
      A(3,3) = A(2,3)
      A(2,3) = T
c9 = ABS(T) > 0
      A(3,3) = A(3,3) + A(3,3)*T
c10 = ABS(A(2,2)) > 0
c11 = ABS(A(3,3)) > 0

```

Figure 4. Augmented Code for Example Program

hand, the two tests shown in Figure 3, denoted as  $T=0$ , are taken to be false on each encounter, and the value  $C_3$ ,  $C_4$ ,  $C_5$ ,  $C_8$  and  $C_9$  will all test positive if the false branches are to be taken. (Since it is only required that  $T$  be non-zero, the  $C$ 's could also be chosen to be negative, but the analysis is tailored around positive valuations.)

It is, of course, possible to express the  $C$ 's to explicitly relate them to the input data. This was done by Miller and Spooner threading back from the predicate, where variable is defined, through intermediate assignments to the original input data. This is fairly simple because the program is straight-lined. The technique is illustrated by a detailed analysis of the auxiliary variable,  $C_7$ . In terms of program variables

$$C_7 = \text{ABS}(A(3,2)) - \text{ABS}(A(2,2)),$$

and these can be traced through the calculations and assignments as follows:

substituting for  $A(3,2)$  and  $A(2,2)$ ,

$$C_7 = \text{ABS}(A(3,2) + A(3,1)*T) - \text{ABS}(A(2,2)^0 + A(2,1)*T);$$

then, since only  $A(2,2)^0$  is input data (and is marked by a superscript, 0) further backing is required; since  $T = A(3,2)^0$  at this point in the program, and  $A(3,2)^0$  is input data, the expression can be written

$$C_7 = \text{ABS}(A(3,2) + A(3,1)*A(3,2)^0) - \text{ABS}(A(2,2)^0 + A(2,1)*A(3,2)^0);$$

but  $A(3,2) = A(1,2)^0$ ,  $A(3,1) = -A(3,1) / A(3,1)^0$  and  $A(3,1)$  in the numerator is equal to  $A(1,1)^0$ . These and similar substitutions provide

$$C_7 = \text{ABS}\{A(1,2)^0 - (A(1,1)^0 / A(3,1)^0) * A(3,2)^0\} - \text{ABS}\{A(2,2)^0 - [(A(2,1)^0 / A(3,1)^0) * A(3,2)^0]\}.$$

This is an explicit representation of  $C_7$  in terms of input.

This illustrates the difficulties in attempting to relate logical paths to input data.

Although this process is feasible for simple programs, and in many respects resembles symbolic execution in reverse, it presents the same difficulties accompanying the development of equivalence classes. An alternative is to work forwardly from the input data to valuations of the C's, and their associated predicates. In this procedure, for properly picked input, all of the C's will be positive and the execution path will proceed along the prespecified path.

The new problem is then one of searching for areas rather than solving for points. These may seem to be problems of the same order of difficulty but they are not. In general applications the searching process need not proceed to the same level of definition that the solving process does. An analogy can be made with polynomial evaluation: it is far easier to locate a point where a polynomial is positive, than it is to find a root for the polynomial.

### 3.1.3 Test Techniques for Segment Coverage

To illustrate some of the characteristics of the test techniques employed the problem discussed above is taken in the framework of the flow diagram of Figure 5. The node numbers shown are in a 1-1 relation to the instructions and labels of Figure 3. DO-loops are easy to identify by the letters E (end) and S (stay), emanating from the end of the loop. The predicates are also easy to identify by means of the T and F letters labelling the exits. The DO1 loop, for example, starts at node 6 and ends at node 9, similarly the DO6 loop starts at 2 and ends at 31. The specified path for the sample problem can be identified in Figure 6. All predicate valuations (that are input dependent) are false except the one inside of the DO1 loop. Both True and False branches were shown to be taken of the nodes 3 and 11, corresponding to the predicates  $K = N$  and  $M = K$ . These are not assigned auxiliary variables but are used to straightline the program; as a result they are permitted either predicate valuation.

Miller and Spooner employ several "objective" functions, generically denoted  $f(C_1, C_2, \dots, C_m)$ ; each has the property that  $f > 0$ , when one or more of the C's is negative, and  $f > 0$  when all of the C's are positive. As an example, the function

$$F(C_1, C_2, \dots, C_m) = \min(C_1, C_2, \dots, C_m)$$

would serve for that purpose.

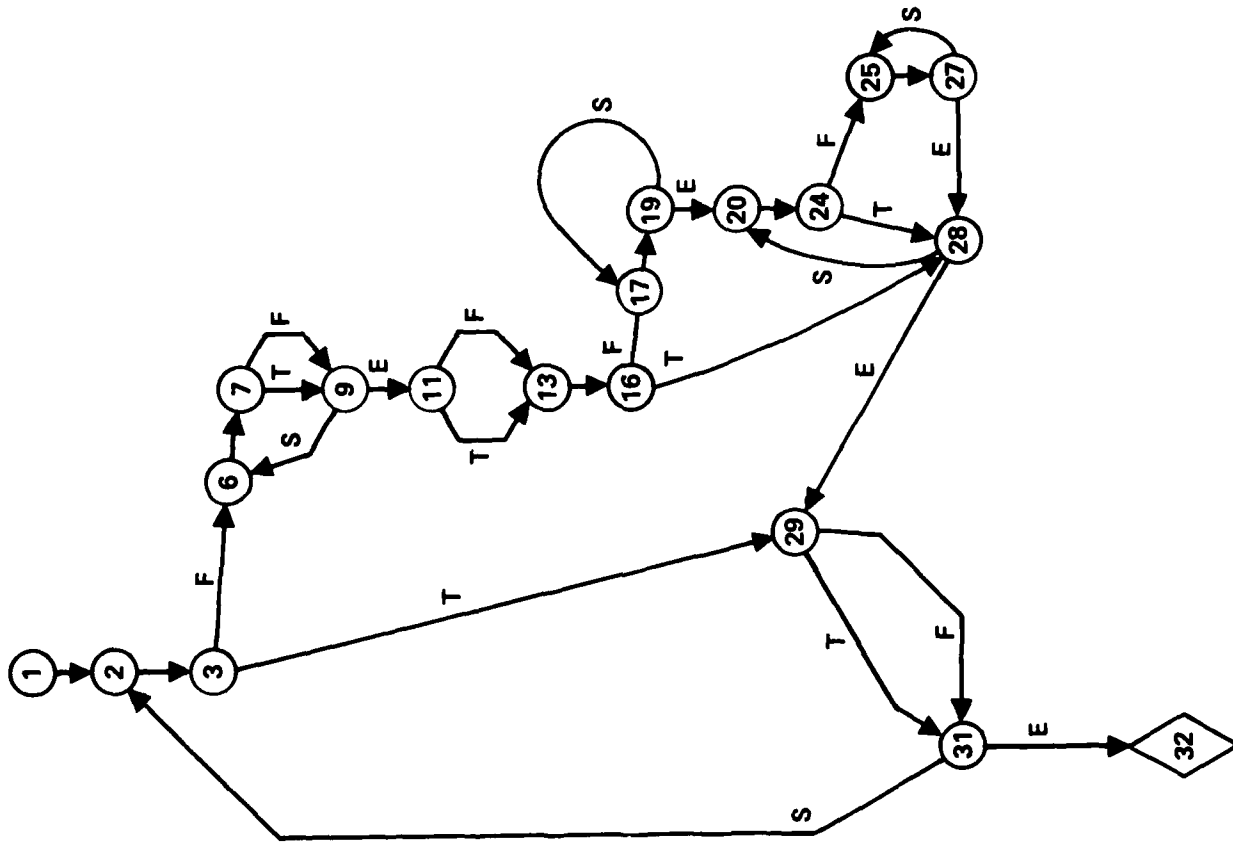
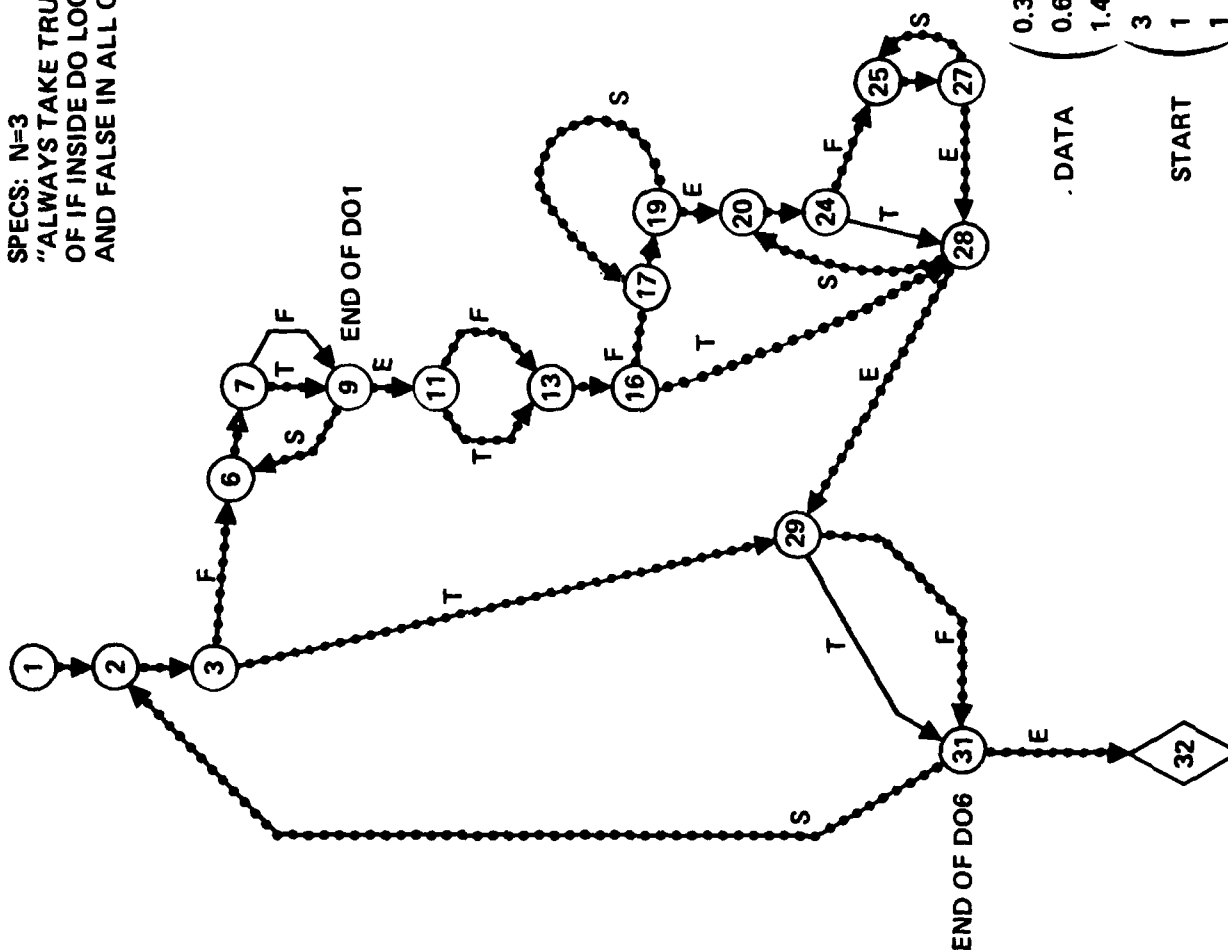


Figure 5. Flow Diagram of Miller and Spooner Example

SPECS: N=3  
 "ALWAYS TAKE TRUE  
 OF IF INSIDE DO LOOP 1,  
 AND FALSE IN ALL OTHERS."



	0.3857	18.62	1.0
DATA	0.6268	-13.865	1.0
	1.439	1.0	5.0
START	3	1	1
	1	4	1
	1	1	5

Figure 6. Selected Path Through Example Program

The problem at hand, then, becomes one of searching over the input data space for values where  $f$  is positive for the specified execution track. In the example problem, Miller and Sponner start the search with a "randomly" chosen matrix

$$A_0 = \begin{bmatrix} 3 & 1 & 1 \\ 1 & 4 & 1 \\ 1 & 1 & 5 \end{bmatrix}$$

which produces  $f = -2$

Using direct search methods, they derive a data set

$$A = \begin{bmatrix} 0.3857 & 18.62 & 1.0 \\ 0.6268 & -13.865 & 1.0 \\ 1.439 & 1.0 & 5.0 \end{bmatrix}$$

which makes  $f=0.2411$ . According to Miller and Spooner, this is accomplished in less than 1 second of CPU time on an IBM 370/168. The resulting coverage of the program is indicated in Figure 6. Because of multiple passes over some portions of the program, depiction is less than perfect. The specified path, however, is achieved by the data.

The usefulness of this procedure is best appreciated when used in conjunction with a combination of the "random" drivers, suggested in the earlier work, augmented by constructed cases. The latter cases, are designed to "fill-in" for data that were taken so infrequently by random numbers that they make the former process uneconomical.

For illustrative purposes, the initial input data is taken as the "random" matrix, used by Miller and Sponner to start the process. For this matrix as input, the FALSE branch out of 7 is taken at least once. Thus, the "random" start exercises a path segment which the "optimum" data does not.

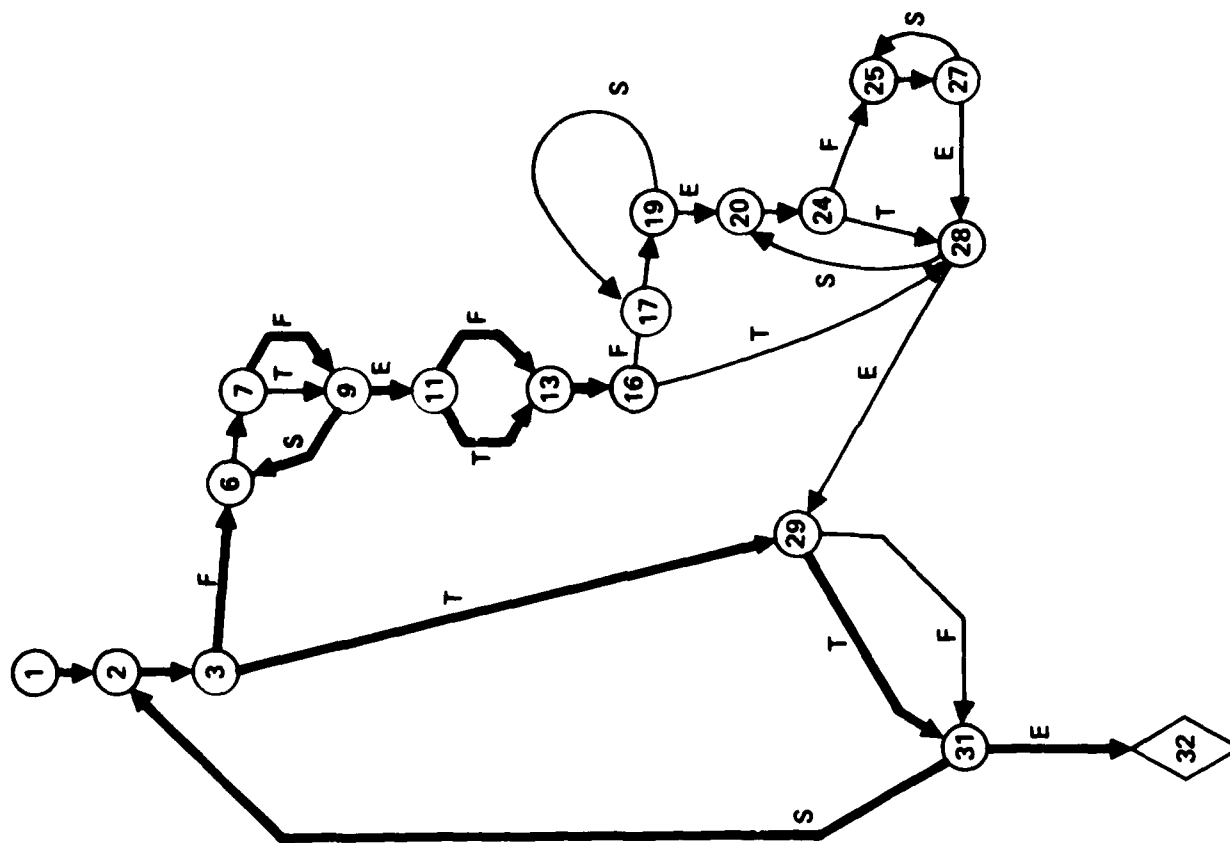
The predicates  $T=0$  are not true unless the zero-valued matrix elements. If the 3x3 zero matrix is used for data, all tests  $T=0$ , as well as the final  $A(K,K)=0$  are true, and the constructed case produces the execution track shown in Figure 7.

Additional tests for programs can often be suggested by some built-in symmetrics in the problem. Thus, for a short problem it can be generally assured that when input data is permuted, the resulting execution tracks will be different. When a polynomial solver is employed it is well known that a set of symmetric relations, involving the roots, define the coefficients of the polynomial. Further, there are relations between the coefficients of a polynomial and the polynomial whose roots are shifted, squared, and inverted. In the present instance of a matrix triangularization, the interchange of two rows can be expected to cause a different response.

As a matter of interest, when the matrix obtained by the optimization process is used with its 1st and 2nd rows interchanged, the resulting track is shown in Figure 8. The False branch out of node 8 is taken on the first entry, and the True branch on (one or more) subsequent passes. (In the particular sequence of tests employed, there is nothing new added by this test).

For the simple problem illustrated here, all segments of the program are tested by the three cases consisting of the starting "random" matrix, the zero matrix, and the matrix obtained by the optimization procedure.

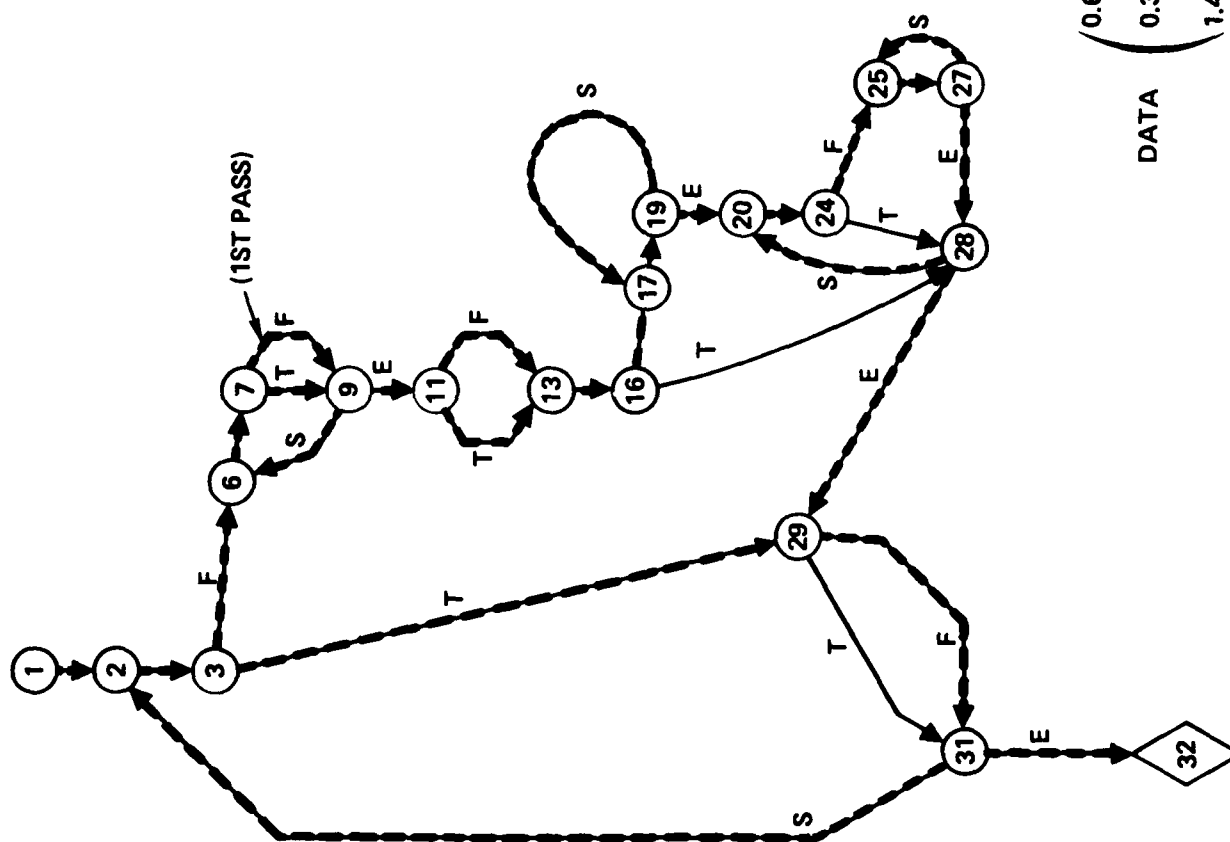
The method suggested in the illustration leads to extensions of value to the general problem of exhaustive testing. As noted in earlier work, the problem of testing a program, only to the point where every instruction and every branch has been executed, is generally a computationally small enough problem making it feasible for almost any program. This is true basically because, for a minimum with  $k$  predicates (two-way), there are no more than  $2k$  data points required to "test" the program in this way, whereas there are as many as  $2^k$  differential logical paths (many more if loops are permitted).



DATA

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Figure 7. Response to Zero Matrix



	0.6268	-13.865	1.0
DATA	0.3857	18.62	1.0
	1.439	1.0	5.0

Figure 8. Response to Data Formed by Interchanging 1st and 2nd Rows of Original Matrices

For the general testing problem, a sequence of random numbers or vectors may be used to develop a set of tracing vectors whose components represent the Boolean valuations of the C's. These runs would generally be both inexpensive and, because they are the first to be employed, would be of high yield. After a reasonably large set of random numbers have been run, the set of associated vectors (as distinct from the values of the auxiliary function exemplified by f in the above discussion) can be examined.

Except for cases where predicates involve equality between expressions involving program variables, the vectors can be collected on the basis of component comparisons. Thus, if there are both zeros and ones\* in the first component position, the testing has "exhausted" the cases provided by the first predicate.

A simple sorting procedure will identify unexercised branches. In case specific predicates are not represented by both "true" and "false" values, the process described above can be used to search for data that will force the program in the desired way. Should there be neither valuation, the same general procedure can be used initially.

It is possible in this scenario, that the so-called "scaling problem", a result of non-common scales on the variables involved which tends to confuse some optimization problems, can be used to advantage in the case of a search for data to exercise a specific branch. For example, if a variable associated with a predicate is more sensitive by multiplication or division of appropriate factors employed in its definition, then strong responses will occur with only small changes in input. A sequence of applications of such factors to each localized variable would, probably, produce good coverage.

#### 3.1.4 Test Techniques for Track Level Coverage

The major use of the above technique is in establishing exhaustive tests for a given program package. The utility as a software metric is clear. As

\*A blank would indicate no test.

noted in Reference 1, one quality of software having universal appeal, is the degree to which a problem has been tested. Ideally this would be measured in terms of the ratio of the number of logical paths executed by all tests performed on the package, to the total number of paths present. However, the latter is almost never known, and there are many non-realizable paths which are not apparent; even the realizable ones may not be easy to enumerate. Thus the more easy to obtain ratio is a substitute.

Reference 1 describes the method of estimating the total number of tracks realizable by random numbers. This method depended on the development of the count of the number of trials between discovery of new paths. An asymptotic limit to the total was then developed on the basis of an algorithm. This technique could be applied to individual branches or to any selected set of branches. Some measure of the degree to which a program has been tested may be developed from the combination of the yields obtained by using constructed cases and from application of random numbers. In specific production-type applications, studies of so-called impossible pairs may be made but for development of a universal metric, such a fine-grained investigation is not warranted.

In order to automate the track-level testing procedure several modifications to the APTS were made and a post processor of data was programmed.

First, the algorithm which obtains the estimated number of tracks through a program obtained by using random numbers as program drivers was programmed as part of a post processing routine. This problem had been solved in principle, but implementation of it heretofore had been effected by the tedious process of desk checking segment usages against all past usages.

The selection of random values for the input variables (real or integer) provides the set of values for one run. The procedure employed for estimating the number of tracks that will be exercised requires a number of executions and comparisons. In the automatic version, the track that accompanies one input data (random) selection is identified in terms of a zero or one assignment to the arbitrarily ordered set of segments which comprise the list of

segments: a zero for nonusage and a 1 for one or more usages. (Two paths which differ in their nonzero counts of the usages of segments, or in their order of execution, are considered to have the same track).

In the implementation of the estimation process, the above outlined initial portion is followed (in the postprocessor) by a routine which compares the sequence of binary n-tuples (one "ordinate" for each program segment) in order to accomplish two things:

A. Establish whether a newly examined track is the same as some track earlier examined, effected by comparing the n-tuples ordinate by ordinate against all previously taken tracks,

B. Marking the trial number of the current track by a zero or 1 in accordance with the results of the comparisons, a zero if an "old" n-tuple has been found and a 1 if the examined track is new.

The data for the estimation procedure consist of the pattern of 0's and 1's obtained in the above comparisons. The primary observable consists of the total trials between adjacent 1's. These spacings between 1's are reported as  $X_1, X_2, \dots, X_n$  and represent the difference in the indices representing trial numbers:  $X_1$  is the separation between the first trial number (by definition, the first trial results in the first new track) and the trial number which produces the second new track (usually this separation is 1 because of the high likelihood that a new data set will produce a different track);  $X_2$  is the separation between the third and second new track, etc.

With data  $X_1, X_2, \dots, X_n$  obtained by running the program over T trials, the number of new tracks is estimated from the equation

$$\sum_{i=1}^n = \frac{1}{N-(i-1)} = \frac{nT}{NT - \sum_{i=1}^n (i-1)X_i}$$

where N is the unknown,  $X_i$  are as defined, T is the total number of trials and n is the number of  $X_i$  employed.

The augmented version of PTS achieves this entire process of comparison and estimation automatically.

## 3.2 APPLICATIONS

### 3.2.1 Air Force Logistics Model--ORLA

In order to avoid the algorithmic-type programs previously studied, programs which are more typical of those encountered in the field were reviewed, specifically the Air Force Logistics programs were reviewed. Inspections of several programs were made for the purpose of selecting a useful candidate for coverage testing. A review of the MOD-METRIC model revealed a very complex program which would have provided an excellent candidate because of the diverse modes which can be exercised. However the fact that documentation of the FORTRAN program is almost non-existent in the mid-levels of documentation (between the overview, on the one end, and inserted comments, on the other), the program was passed over. The LSC (Logistics Support Cost) model was not selected because it consists of a set of rather simple algebraic formulas. Another model, LEM (Logistics Effect Model) is not yet widely known in the Air Force, and primarily was eliminated for that reason. The Air Force LCOM (Logistics Composite Model) was investigated and while its basic or underlying language is FORTRAN, it has a language of its own and is not therefore suitable for analysis. Another difficulty with LCOM is that production runs with that model would cost far in excess of any contemplated expenditures for the testing task which was planned. This is so because the model relies on simulations with an underlying SIMSCRIPT II program, to produce Monte Carlo based statistics of operational parameters. The program with greatest potential among those investigated is commonly called ORLA (Optimum Repair Level Analysis). The particular version employed was written by O. R. Johnson of Warner-Robins Air Force Logistics Center.

ORLA employs costs associated with the acquisition, logistic support, and replacement, or airplane subsystems. Three options are generally considered in an ORLA analysis: discard at (suspected) failure of the subsystem; repair of the failed subsystem at the base (home airport), or repair at an Air Force depot (generally supporting several bases). Some 11 different cost components are involved for the latter two options, while 3 cost components comprise the discard option total. Although computations are not complex, and, indeed, the cost components are simply algebraic formulas, the so-called sensitivity analysis presents some interesting complexities and decisions.

The aim of this sensitivity analysis is to determine (to the nearest 1% of the baseline value) the point at which the nominal decision, derived from the baseline values, will be reversed. This is accomplished for any choice from the 17 different input factors, and it provides, as the name indicates, a measure of the sensitivity or stability of the decision in the face of possible changes in or misestimation. The sensitivity analysis is flow-charted in Section 3.2.1.1 where the application to the ORLA program is illustrated.

#### 3.2.1.1 Segment Level Coverage of ORLA

The main ORLA program consists of 488 lines of FORTRAN code (each branch of branching instructions are counted). Briefly the components of ORLA can be described by the following: Initialization (about 15 instructions); Read Constants (64); Compute Failure Rate (59); Computation of Aerospace Ground Equipment Usage (66); ORLA Variable Identification (34); Economic Analysis (33); Write Summary (15); Computation Routine (58); Rank Economic Values (22); Sensitivity Analysis (93); Write Repair Summary (12). (In addition three peripheral and non-essential subroutines are included in the program: two are merely messages for the user in case he requires explanations of the program, the third is set of error messages in case of inconsistencies in the data. These subroutines are not included in the discussion which follows.) A listing of an APTS-augmented ORLA is given in Appendix A.

To drive the basic ORLA a total of 54 variables are employed. These variables provide descriptions of all the logistics involved in acquiring, shipping, repairing, maintaining, and resupplying an aircraft subsystem. Included are variables which represent overhead, such as, training of maintenance personnel, management of inventory, and facilities. The 54 variables are divided into 2 main classes. First a set of 36 variables describe the rates which hold or are projected to hold for the time of the analysis, the force size and deployment scheme, labor and material rates, and so forth. In addition to these, a second class bears directly on the item or subsystem analyzed (ORLA'd); there are 17 variables in the class and they describe, cost and weight of the subsystem and its parts, repair time, and the documentation, training, and special facilities which are required for the item.

In addition to these basic variables there are 10 additional variables which are derived from intermediate computations which rely either on keyboard entry (of parameters relating to the MTBF) or on sharing of resources by several items (AGE or test equipment which is employed or several different subsystems of the aircraft for example). The reason for identifying them with the input variables is that they also can be subjected to the sensitivity analysis.

As noted earlier the ORLA program employs the input values associated with a given item and computes the costs which would be incurred under the three options (discard, repair at base, repair at depot). On the basis of the three ranked costs the optimum or least cost repair level can be determined. Although the numerical values of the costs of the various components of cost are printed out and an indication of the assurance or firmness of the decision which the program makes can be made from these magnitudes, a better measure of the firmness of the decision can be made by use of sensitivity analyses. Each run a set of up to 10 user-selected variables can be identified for use in this analysis. As noted before, the primary purpose is to determine, from variations in the costs due to changes in the selected variable, the point (a percentage of nominal value) where the decision based on nominal or baseline costs is reversed. This is determined to the nearest percentage on the range 20% to 500% (1/5 to 5 times nominal). Should no change in decision occur over this range, the decision is clearly stable with respect to the variable inspected.

Certain variables are known to affect certain options more than others and a user wishing to test for coverage could be guided by this a priori knowledge. Some of this kind of knowledge is also used in the construction of cases which are discussed here. This is counter to the mode which would be used in the final testing scheme where it is assumed the user is unaware of the relationship between input and any particular program segment. In the final version each variable would be varied at random to provide an initial coverage; subsequent coverage would be initiated by a specification of a program path or track, then continued by invoking a search procedure on the input data, and hopefully consummated by an identification of a point which

produces an execution which includes selected path or track. Because the status of the study has not progressed to the point where automatic insertion and data generation are possible the procedure used in the example relies on knowledge of the program.

It is cumbersome to illustrate the usage of APTS on the entire ORLA program, but a good indication of the way APTS can be applied in static analysis can be provided by use and inspection of a compact portion of the listing. In Figure 9 is a flow chart of the portion of the program called Reversal Analysis. This is used in part of the sensitivity analysis to compare and rank the costs of the three options. For convenience the ORLA program with segments identified comprises Appendix A.

Application of APTS in static composition of segments from the coding of the above identified program portion is effected by first numbering the instructions as shown in Figure 10. This shows the numbering in the leftmost column and these are associated with the instruction on the right. Labels shown correspond to the original listing and are employed in the flowchart of Figure 9. Thus 396 corresponds to the labelled (215) instruction, `JSEN(1)=KDT`, at the top of Figure 9, 415 corresponds to the predicate, `NUMK(1)-NUMJ(1)=0`, which appears just after the labelled 310 CONTINUE instruction in Figure 10. The APTS segmentation of the program in the above described region is shown in Figure 11.

It is important to note that in most cases the segments are made up of several of the PTS segments defined in Reference 1. Those segments were truncated by labels, GOTO's, etc. Several other points require explanation. First the segments identified with the bracket/parenthesis, start with an instruction number which is either the start of the program or subroutine, or a predicate (IF statement in most cases), the remainder of the numbers in the sequence denote the instructions which will be executed in sequence, the end of the sequence of numbers is identified by a number corresponding to a predicate or branch point. Thus  $T_1$  starts with the labelled instruction 396, then in turn by 397, 398, 399, 400 and ends with 401. The instruction 401 is an implied predicate, `DO LOOP END=TRUE`. If the predicate is true the next segment taken is  $T_3$  which describes passage from the D0210 loop to

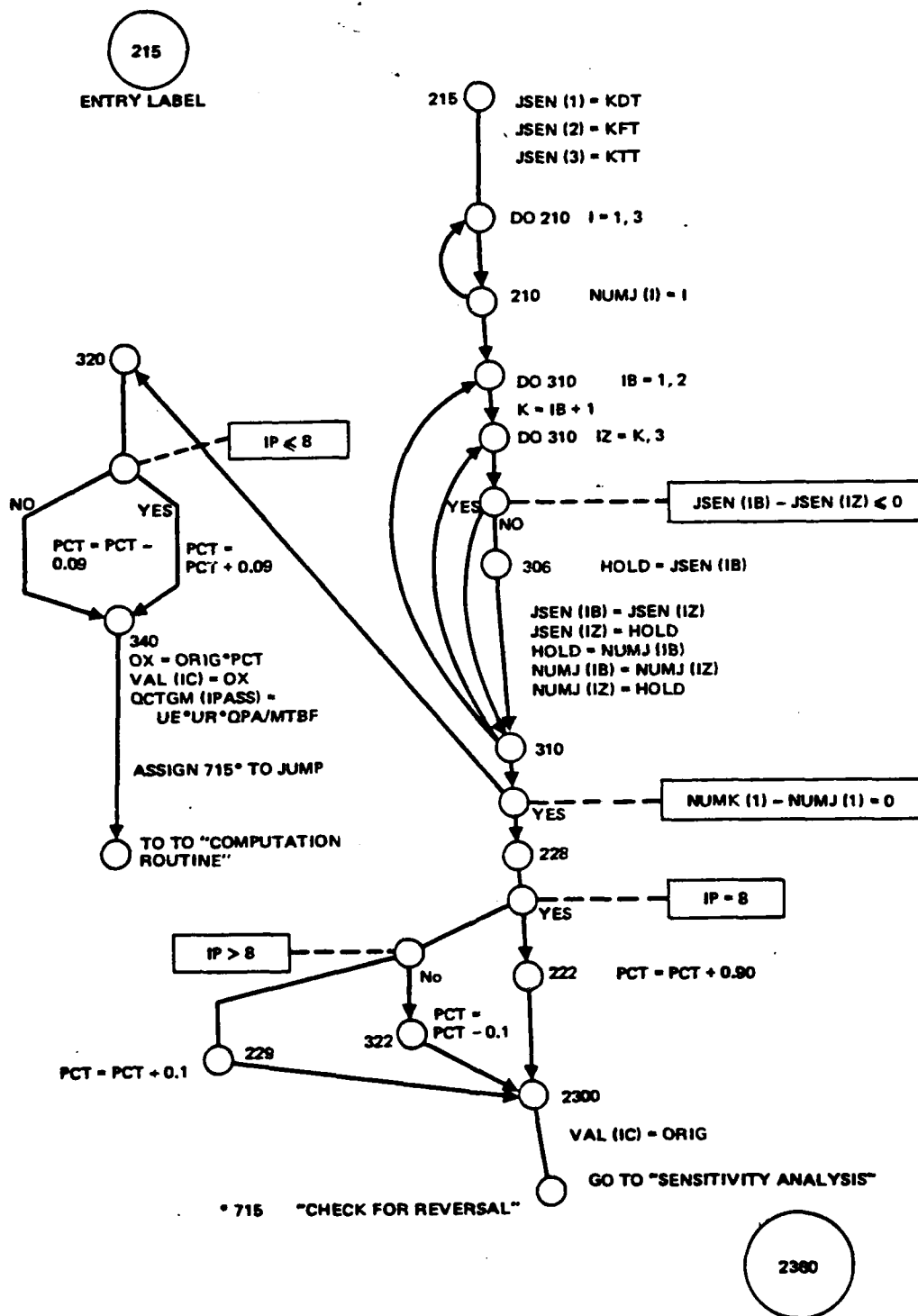


Figure 9. Reversal Analysis

# REVERSAL ANALYSIS

```

396      215 JSEN(1) = KDT
397      JSEN(2) = KFT
398      JSEN(3) = KTT
399      DO 210 I = 1,3
400-401    210 NUMJ(I) = I
402      300 DO 310 IB = 1,2
403      K = IB +1
404      DO 310 IZ = K,3
405      305 IF (JSEN(IB)-JSEN(IZ)) 310,310,306
406      306 HOLD=JSEN(IB)
407      JSEN(IB) = JSEN(IZ)
408      JSEN(IZ) = HOLD
409      HOLD = NUMJ(IB)
410      NUMJ(IB)=NUMJ(IZ)
411      NUMJ(IZ)=HOLD
412-414    310 CONTINUE
415      IF (NUMK(1) = NUMJ(1)) 320,228,320
416      228 CONTINUE
417      IF (IP-8) 322,222,229
418      222 PCT = PCT-.90
419      GO TO 2300
420      322 PCT = PCT-.1
421      GO TO 2300
422      229 PCT = PCT + .1
423      GO TO 2300
424      320 CONTINUE
425      IF (IP-8) 375,375 , 360
426      375 PCT =PCT + .09
427      GO TO 340
428      360 PCT = PCT -.09
429      340 OX = ORIG * PCT
430      VAL(IC) = OX
431      QCTGM(IPASS)=VAL(31) * VAL(32) * VAL(46)/VAL(56)
432      ASSIGN 715 TO JUMP
433      GO TO 100

```

Figure 10. APTT Numbering for Program

# MDAC SEGMENT XLATOR

T <sub>1</sub>	[396-401)
T <sub>2</sub>	[401,399-401)
T <sub>3</sub>	[401-405)
T <sub>4</sub>	[405,412-413)
T <sub>5</sub>	[413,404-405)
T <sub>6</sub>	[413-414)
T <sub>7</sub>	[414,402-405)
T <sub>8</sub>	[414-415)
T <sub>9</sub>	[415,424-425)
T <sub>10</sub>	[425-427,429-433,304-319)
T <sub>11</sub>	[425,428-433,304-319)
T <sub>12</sub>	[415-417)
T <sub>13</sub>	[417,420-421,460-461)
T <sub>14</sub>	[417-419,460-461)
T <sub>15</sub>	[417,4220-429,460-461)
T <sub>16</sub>	[405-413)

Figure 11. APTS Segment Identification

the D0310 loop, and continuation to the next predicate which is an explicit predicate, JSEN(IB-JSEN(IZ)=0 . If the predicate is false, the segment  $T_2$  is executed, with an initial 401, the entry or reentry into the D0210 loop at 399.

Because of the selection of only a portion of the program, some segments shown in Figure 11, such as  $T_{10}$ , list instructions which are outside the range of those shown in Figure 10. The explanation for  $T_{10}$  which will be given here should serve for others as well.  $T_{10}$  is made up of (425-427, 429-433, 304-319), and it is the last group that is out of range. Instruction 433 is a GOTO 100 instruction and the APTS number 304 corresponds to the label 100, which is the start of the so-called Computation Routine. This routine computes cost components for the three options and the 304-319 segments is the initial segment of that routine. A "return" to the portion which is displayed in Figure 9 is at the end of the Computation Routine (at APTS number 355-not shown). There is one entry point from the Computation Routine and that is at  $T_1$ . So far as the local analysis is concerned  $T_{10}$  joins to  $T_1$ . The original set of segments can be tailored so as to exhibit only local connections by the above method. So far as the illustration of technique is concerned, however, there is no need to work at that level of detail.

Figure 9 shows the two major exits: computation routine (label 100 and APTS number 304); sensitivity analysis (label 2360, APTS Number 384). The program was initially driven with a set of standard elements for one of the Air Force's aircraft types and an imaginary subsystem. The program's pre-selected variables were used in the sensitivity analysis (these correspond to the size of the force being fitted, the number of hours per month the aircraft will be used, the repair manhours, the unit cost, the Mean Time Between Failures, the cost of depot AGE, and the cost of base level AGE). The initial data exercised the segments listed in Figure 11 as follows.

<u>Segment</u>	<u>Number of Executions</u>
$T_1$	118
$T_2$	236
$T_3$	118

<u>Segment</u>	<u>Number of Executions</u>
T <sub>4</sub>	241
T <sub>5</sub>	118
T <sub>6</sub>	236
T <sub>7</sub>	118
T <sub>8</sub>	118
T <sub>9</sub>	5
T <sub>10</sub>	3
T <sub>11</sub>	2
T <sub>12</sub>	113
T <sub>13</sub>	28
T <sub>14</sub>	4
T <sub>15</sub>	81
T <sub>16</sub>	113

For this arbitrary set of data all explicit and implicit predicates were exercised. This complete (local) testing was fortuitous in a sense, for in three successive runs with other data T<sub>10</sub> was not exercised, while T<sub>9</sub> and T<sub>11</sub> were not exercised in one case.

The static aspects of APTS are well illustrated by the foregoing. The dynamic aspects can be illustrated by the results from four data sets. The first or nominal is the set identified above, the second maintained the same standard elements and changed one item parameter, the unit cost (from 3600 to 36). The third restored the unit cost to the original value and changed one standard element, depot labor rate (from 12.44 to 1). The fourth changed the turnover rate from 0.15 to 15.

Results over the entire 113 segments of the program show that the initial choice of data was indeed exceptional, since 96.63% (86 out of 89) of the segments exercised by the four segments were exercised by the initial set.

The change in cost by a factor of 100 (the second case) exercised two segments not exercised by the first set and these correspond to predicate branches caused by the re-ranking of the costs of the three options (discard would be the least expensive). Similarly for the fourth set, a reversal of the costs of depot and intermediate repair is effected by the extreme value chosen for depot turnover.

Examination of the 113 segments comprising the ORLA program, shows that 24 segments were unexercised by the four simple cases. But, of these, 13 depend on choices which are prompted by the program; that is they are yes/no responses to questions concerning choices as to whether the user wishes to correct an entry, whether he wishes an explanation, whether he wants to run a batch of several items, etc. In some cases these choices reflect into the substance of the program and in others they stimulate isolated calls and returns without exercising any computations. Of the 11 segments which remain, all but four can be exercised with data.

As a very simple and brief explanation of the actual technique used for constructed cases, and as a useful means of discussion of the automated version of the process, the predicate,  $EOQ < A$ , involving the two program variables EOQ and A will be discussed.\* The APTT post-processor tally usages of the entire program shows that the true branch of this predicate is taken on every encounter (1143 passages in the 4 cases). The code contiguous to the predicate shows that the true branch corresponds to the inequality:

$$4.4 \sqrt{A} < A$$

or

$$19.36 < A$$

\*These variables occur in the Computation Subroutine and represent Economic Order Quantity (EOQ) and a "Pipeline" content (A).

Again by use of other parts of the code, it is established that

$$A = 12 \cdot V_{45} \cdot V_{48} V_{31} \cdot V_{32} \cdot V_{46} / V_{56}$$

where the V's are all input variables.

Thus the choice  $V_{45} = 0$ , among many others, will cause the false branch to be taken.

It is well to note that in the contemplated scheme, random numbers would be used over convenient ranges for all of the input variables, and, in this case, the probability of producing an A value less than 19.36 would be extremely high. Thus it is highly likely that the case investigated here would not have arisen in the context of an unexercised branch at a corresponding stage of testing, and in fact, when 100 cases were run this branch was indeed executed.

Should a similar predicate branch be untested after an initial set of data runs, the following procedure would apply. The augmenting program variable  $C = EQQ - A$  would be inserted at the predicate site during the APTS pre-processing. During each pass the value of C would be evaluated (in combination with other inserted augmenting variables at other sites of predicates). Variations on the input data would be made according to a search scheme until a point is reached where all augmenting variables have the desired sign - in the present case, C must be positive.

The more extensive test of ORLA comprised a run of size 100. Several interesting problems arose in the process of obtaining these runs.

Most of these problems concerned character string inputs. To test in a random way, the variables of ORLA, the user must become somewhat familiar with the sites where meaningful input is done, and what type of input is expected. There are five types of input required by ORLA:

- 1) real variable containing either "yes" or "no"
- 2) real variable containing real values

- 3) integer variable containing integer values
- 4) double precision variable containing an a8 string
- 5) double precision variable containing one of sixty-four possible a8 string names

Because FORTRAN allows character strings to appear in all data types, trying to recognize inputs and generate random values for them causes a major problem. After the sites for inputs from the user were established they were replaced by a call to a hand-generated input routine of the proper type.

It was decided to run one hundred test cases using the random inputs as test values. The ORLA source program was pre-processed by APTS and compiled, then linked to the random input routines. One hundred executions of the instrumented program followed. For each execution, an output report was generated by ORLA and a post-processor report was generated by APTT. A log was also kept for each test case run. There were five types of run-time errors that were detected by the FORTRAN run-time library.

- 1) Floating point divide check
- 2) Floating point overflow
- 3) Square root of a negative number
- 4) Integer overflow
- 5) Illegal character in data

Each of these errors is not an expected output of the ORLA program. At this point, an interesting point should be made about program testing. To facilitate the testing of computer programs where there is a possibility of run-time errors, either fatal or non-fatal, there must be a mechanism for gathering the statistics that have been collected up to the point of the error. Fortunately the DEC-10 operating system has such a facility.

After one hundred test case executions, only four segments failed to be executed:

- 1) Segment 86 [457,461), To execute this segment there must be a premature end of file on FORTRAN logical unit IWORK2. This appears to be impossible because the loop which reads the data from this file is controlled by a variable that is incremented for each write to IWORK2. (See line 277.)
- 2) Segment 96 [199-204), To execute this segment the variable ITAGE must be less than two and the AGE SUMMARY option must have been selected.
- 3) Segment 98 [187,200-204), This segment appears to be impossible to execute under all input values. If variable ITAGE is greater than two and the AGE SUMMARY option has been selected then the loop from statements 143 to 199 would be exited at statement 169 before segment 98 has a chance to be executed.
- 4) Segment 104 [64-64,62-64), This segment was not executed due to the restricted range of values selected for random input to variable IT. If the range had been expanded from (0,10) to (0,11) then segment 104 should have been hit.

Thus segment coverage by the 100 test cases was essentially complete. Two of the segments are apparently impossible to execute, and two require user options which could be taken but were not. The comprehensiveness of the random number testing is clearly demonstrated in this example.

#### 3.2.1.2 Track-Level Coverage of ORLA

The 100 test cases which were used for the segment coverage testing were also employed in the analysis of track coverage. This number turns out to be inadequate for this purpose but the difficulties which were described in the previous subsection proscribed any attempt at exhaustive testing. The fact is that 98 out of the hundred tracks generated were unique. This relatively simply formula-oriented program requires a test sample of at least 100 different

runs and as indicated below the probability is high that several hundred or several thousand may be required. This is in stark contrast to the fact that almost all segments have been covered.

For ordinary programs there would not be any problem for generation of random input is "from the top" and can be inexpensively provided, whereas, for the interactive ORLA, requests for input must be responded to by on-line monitoring, resulting in constant attention and manual input of information.

Nonetheless, the procedure of track estimation can be well illustrated by considering the initial segments of the ORLA, and sequentially increasing its size from 15 to 75 in steps of 15. Estimates are made on these segments to produce trend data.

In Table I (two parts) the segment usages of the complete ORLA program are shown. This program consists of 113 segments in the main program. These correspond to the first 38 octal numbers to the left of the arrow between the 38th and 39th number. Those to the right of the arrow represent subroutines. Each of the first 37 octal numbers represents usages of three consecutive segments. An octal digit of 5 in the first position indicates, for example, usage by the 1st and 3rd segments and non usage of the 2nd, a 7 indicates usage by all three segments. This coding is continued, each representing 3 consecutively listed segments. The 38th digit represents a mix of the 112th and 113th segment of the main program and the first segment of the first subroutine (which is immaterial).

It is noteworthy that the subroutines of the program have apparently or probably been fully tested at the track level since the octal numbers (in the 39th through 41st columns), 375, 777, 775, 377, 335, 001 appear to comprise all tracks, with no new occurrences past the 36th run number.

As noted above the number of runs made could not serve to test the entire (113 segments) program. But it is interesting to analyze the problem from the bottom up.

Table I (Part 1). ORLA Segment Usage Versus Trial Number  
(Page 1 of 2)

QLOOP : 100

↓

1 :	1	575677777777777777577777617755660757673750000000
2 :	1	775757777777777777377777777775740000157667770000000
3 :	1	77367777777777777737777777777757000700077750000000
4 :	1	573677777777777777377577737617753760700063770000000
5 :	1	7777777777777777777777777777740000100067750000000
6 :	1	5716777775777377577737612751660757663350000000
7 :	1	7776777777777777773776177000007753000700067770000000
8 :	1	5776777771777377757777617751000700063750000000
9 :	1	5717777777777777773777777777736740000155663350000000
10 :	1	77577777777777777737757777617740000157667770000000
11 :	1	57577777777777777737777777777740000157663350000000
12 :	1	57735777757703736177000003640000100063750000000
13 :	1	7716777771777377737777537755000757777750000000
14 :	1	57567777777777777737777777777757765757673750000000
15 :	1	5716577777777777773776177000005753000755663750000000
16 :	1	57577777757777776177000007740000155663750000000
17 :	1	5757777777777777774377777777777740000157663750000000
18 :	1	7716777777777777777757777617755000757677750000000
19 :	1	57777777777777777737777777777740000100063750000000
20 :	1	571677777577777777401777755015000757673750000000
21 :	0	77777777777777777777777777777740000100067750000000
22 :	1	57177777777777777743777777777757740000157663750000000
23 :	1	776263777577743776177000002653000700067750000000
24 :	1	575657777177733776177000003751000757663750000000
25 :	1	5776577775777377757777617753000700063750000000
26 :	1	5757777777777777773776177000007740000157663750000000
27 :	1	776277777577743777401777614653000700067350000000
28 :	1	573677777577777777777777773757000700073750000000
29 :	1	77477777757773776177000003740000157667750000000
30 :	1	5716777777777777773776177000007757000757673750000000
31 :	1	777777777577767777777777777740000100067750000000
32 :	1	7776777777777677757777617751000700067750000000
33 :	1	5756777775777677757777617753000757663750000000
34 :	1	5736777777777777773776177000006753765700063750000000
35 :	1	5776777777777777773776177000002753760700063750000000
36 :	1	5706777777777777774377777777773753760757660010000000
37 :	1	77177777777777777777757777617740000157667750000000
38 :	1	771677777777777777777777773755000757677350000000
39 :	1	577677777777777777377777777757755000700073750000000
40 :	1	57375777757776777757777617740000100063350000000
41 :	1	773677777777777777377757777617755000700077750000000
42 :	1	573677777777777777377757777757751065700063750000000
43 :	1	77177777757774377757777557740000157667750000000
44 :	1	7756777775777777777777777737755000757677750000000
45 :	1	5776777777777777773776177000002751000700063350000000
46 :	1	775777777177733776177000007740000157667750000000
47 :	1	575677777777777777777777777757000757673750000000
48 :	1	57077777777777777737777777777740000157660010000000
49 :	1	57367777777777777737777777777751065700063350000000
50 :	1	77767777757703736177000003751000700067350000000

Table I (Part 2). ORLA Segment Usage Versus Trial Number  
(Page 2 of 2)

51 :	1	773677777177733776177000007753000700067750000000
52 :	1	573677777777377777777777755660700073350000000
53 :	1	771677777777377777777777755000757677350000000
54 :	1	77577777777733776177000007740000157667750000000
55 :	1	575677777777377757777617751000757663750000000
56 :	1	7717777757773776177000007740000157667350000000
57 :	1	5716777777773777777777753000757663750000000
58 :	1	775777777777777757777617740000155667750000000
59 :	1	77367777777733776177000007757065700077350000000
60 :	1	77367777577737777777777751660700067750000000
61 :	1	771777777777377377777415740000157667750000000
62 :	1	7756637777733776177000002757000757677750000000
63 :	1	77367777177733776177000003751000700067350000000
64 :	1	57167777177733776177000007757065757673750000000
65 :	1	573677777777617777777773757760700073350000000
66 :	1	5736777757774377757777617755000700073750000000
67 :	1	5736777777773777777777755000700073750000000
68 :	1	773777775777377577737617740000100067750000000
69 :	1	77767777777737757777617751000700067750000000
70 :	1	57767777777777757777617755000700073750000000
71 :	1	57167777577777757777617755000757673750000000
72 :	1	77177777777737777777777740000157667750000000
73 :	1	77767777177737777777777755000700077750000000
74 :	1	57567777777737757777617757660757673350000000
75 :	0	57567777777737757777617757660757673350000000
76 :	1	57567777777777777777737755065757673750000000
77 :	1	571677777777437777777777753000757663750000000
78 :	1	577677775777377577737617757760700073750000000
79 :	1	7757777777773776177000007740000157667750000000
80 :	1	77577777777737777777777740000157667750000000
81 :	1	77777777577777777777773740000100067750000000
82 :	1	7777777777773776177000006740000100067750000000
83 :	1	575777775777777757777617740000157663750000000
84 :	1	571677777777437777777777757000757673750000000
85 :	1	77767377777733776177000007757000700077750000000
86 :	1	577777771777337777777775740000100063750000000
87 :	1	77567777777733777377777417757765757677750000000
88 :	1	57367777777733776177000007755660700073750000000
89 :	1	5776777777773776177000003751000700063750000000
90 :	1	775777775777377577737617740000157667750000000
91 :	1	57777777777777777777777740000100063750000000
92 :	1	77577777777733776177000003740000155667750000000
93 :	1	7716777777773776177000007757165757777750000000
94 :	1	7736777757773777777777755065700077750000000
95 :	1	7736777757776776177000003753000700067750000000
96 :	1	777777777777377777777757740000100067750000000
97 :	1	7777777757777777777777617740000100067750000000
98 :	1	7756777717772776177000007753000757667750000000
99 :	1	5737777777777777777777617740000100063750000000
100 :	1	7776777757773777777777751000700067750000000
ALL :	0	777777777777777777777777757765757777770000000

The initial analysis on the main program was carried out on the first five octal numbers (representing the 15 initial segments of the list of segments shown in Appendix A). It is well to note again that the octal number 57567, or binary 101111101110111 associated with the first run, means that segments 2, 8, and 12 were not exercised and all the rest were exercised. By comparison of the first five numbers of each run with its predecessors the pattern of occurrences of new (partial) tracks can be established. From this sequence the  $X_i$ 's of the algorithm can be established as

$$\begin{aligned} X_1 &= X_2 = \dots = X_{12} = X_{13} = 1 \\ X_{14} &= 2, X_{15} = 4, X_{16} = 4 \\ X_{17} &= X_{18} = 1; X_{19} = 2, X_{20} = 2 \\ X_{21} &= 7, X_{22} = 1, X_{23} = 3, \\ X_{24} &= 4, X_{25} = 4, X_{26} = 14 \\ X_{27} &= 6, X_{28} = 31. \end{aligned}$$

(The Sequence of Boolean symbols is not written because they can be recovered from the  $X_i$ : 13 ones, 1 zero, 1 one, 3 zeroes, etc.)

The ratio  $\sum(i-1) X_i / \sum X_i$  in this case is 20.94, and the tables in Appendix II of Reference 1, indicate that the expected residual track count (by extrapolation) is less than 0.04 (notwithstanding the occurrence of a unique track on the 99th run).

For the first 30 segments (i.e., first 10 octal numbers) the pattern is

$$\begin{aligned} X_1 &= X_2 = \dots = X_{13} = 1; X_{14} = 2, X_{15} = 1; X_{16} = 2; \\ X_{17} &= 1, X_{18} = 4, X_{19} = X_{20} = X_{21} = X_{22} = X_{23} = X_{24} = 1; \\ X_{25} &= 2; X_{26} = X_{27} = X_{28} = X_{29} = 1; X_{30} = 3; X_{31} = 1; \\ X_{32} &= X_{33} = 2; X_{34} = 3; X_{35} = 9; X_{36} = X_{37} = 2; \\ X_{38} &= 4; X_{39} = X_{40} = 5; X_{41} = 7; X_{42} = X_{43} = 1; \\ X_{44} &= 3; X_{45} = 8; X_{46} = X_{47} = 1 \end{aligned}$$

The ratio  $\sum(i-1) X_i/X_i$  is 29.13 and tables for  $n = 47$  in Appendix A of Reference 1, show that this corresponds to a residual count of 10.9 tracks.

In the context of the present illustration this means that there remain 10.9 tracks which will exercise the first 30 segments differently from the way they were exercised in the first 100 runs and which will differ from one another.

A very coarse approximation to the total testing required can be found by multiplying the number of remaining tracks by the mean number of trials between occurrences of the next track as provided by the entry for the MTTF analog to this in the tables of Reference 1. In this case this meantime is about 5.8 so that total testing will require in excess of 63 additional trials (163 in all).

A better estimate can be obtained repeated use of the tables, in this way the stretching out of the MTTF which occurs as new tracks are found can be accounted for. Using the aforementioned tables this estimate for the additional trials  $\hat{s}$  is

$$\begin{aligned}\hat{s} &= 5.6 + 5.7 + 6.0 + 6.5 + 7.0 + 7.7 + 8.5 + 9.9 + 12.0 + 16.6 + 30.1 \\ &= 115.6\end{aligned}$$

where the individual terms are taken from the MTTF column of the tables for  $n = 47$  to 57. The refined estimate is that about 116 additional runs are required.

For the 45 initial segments the separations between new tracks are:

$$\begin{aligned}X_1 &= X_2 = \dots = X_{19} = 1; X_{20} = 3; X_{21} = X_{22} = X_{23} = 1; \\ X_{24} &= 2; X_{25} = X_{26} = \dots = X_{37} = 1; X_{38} = 3; \\ X_{39} &= 1; X_{40} = 2 = X_{41} = X_{42}; X_{43} = 1 = X_{44}; \\ X_{45} &= 2; X_{46} = X_{47} = 1; X_{48} = 2; X_{49} = X_{50} = X_{51} = X_{52} = 1, \\ X_{53} &= 2; X_{54} = 1 = X_{55}; X_{56} = X_{57} = 2; X_{58} = 1;\end{aligned}$$

$$\begin{aligned}
X_{59} &= 2; X_{60} = 1; X_{61} = 3; X_{62} = 1; X_{63} = 3; X_{64} = 1; \\
X_{65} &= 3; X_{66} = X_{67} = 1; X_{68} = 3; X_{69} = 1; \\
X_{70} &= 2; X_{71} = 2; X_{72} = 1; X_{73} = 2; X_{74} = X_{75} = 1.
\end{aligned}$$

The pattern of separations of occurrences, produces an estimate of 95.5 (170.5 total) additional tracks, and a mean time to next new track of only 1.82. The number of trials required to achieve perfection can be approximated by the formula for MTTP for Section 2.2.3.2. In this case  $N = 170.5$ ,  $\phi = 0.00576$ , and

$$MTTP \approx \frac{1}{\phi} \sum_{i=75}^{169} \frac{1}{170 - i} = \frac{1}{\phi} \sum_{k=1}^{95} \frac{1}{k}$$

which can be approximated by the sum of the logarithm of  $n$  and the Euler constant

$$\begin{aligned}
MTTP &\approx \frac{1}{\phi} (\ln 95 + 0.57721) \\
&\approx 890
\end{aligned}$$

Thus 890 additional tests are estimated to be required for a complete test.

For the first 60 segments, the pattern of 87  $X_i$ 's, produces an estimate of the undiscovered tracks of about 359, of  $\phi = 0.00217$ , and the total number of runs required for a fully tested program is about

$$\begin{aligned}
MTTP &\approx \frac{1}{\phi} \sum_{i=87}^{445} \frac{1}{446 - i} = \frac{1}{\phi} \sum_{k=1}^{359} \frac{1}{k} \\
&\approx 300
\end{aligned}$$

The analysis for the initial 75 segments produces 91 separation intervals, with a pattern showing only 8 values of  $X_i$  differing from 1. These are all 2

and occur at the indices 21, 43, 51, 60, 71, 79, 82, 85, and 86. These produce an estimate of  $N = 1108$ ,  $\phi = 0.000857$ . Corresponding is an

$$MTTP \approx \frac{1}{\phi} \sum_{i=92}^{1107} \frac{1}{1108 - i} = \frac{1}{\phi} \sum_{k=1}^{1016} \frac{1}{k}$$

$$\approx 8750$$

Naturally these later estimates are extremely weak with unquestionably extremely large variances. The point with any such estimates is one of determining the status of testing and gross estimates are sufficient.

The preceding sequence of tests clearly indicate by the increasingly large value of MTTP that the testing required is extremely large, probably in excess of 50,000 runs. And this is for track level testing on coverage, not execution sequence coverage.

It is useful to note that the bottom row, denoted all, which shows in each position the "union" of all octal tokens above it in the column, shows segment coverage complete through 85 segments. (86 was noted before an impossible segment.)

### 3.2.2 Comprehensive Testing of Matrix Triangulization Problem

The matrix triangularization example discussed earlier will be reexamined. Directed graphs of the potential program flow, and a few examples of the coverage by random numbers and constructed cases were given in Section 3.1.2. Listings of the MAIN and TRIANGULARIZATION subroutine comprise Figure 12.

Appendix B contains tables and reports of the APTS output for three separate test runs. The reports show the testing coverage provided by using the user-described input routine INROUT. INROUT returns a new set of randomly distributed over the logarithm in the range -2 to 1. The sign of the individual data items is also selected randomly.

```

0- 1      PROGRAM MAIN
      IMPLICIT INTEGER(A-Z)

      INTEGER IP(3)
      CALL 1(4,3),R

      OPEN(UNIT=20,DEVICE='DSK:',FILE='TRIANG.RES',ACCESS='SEQUENT
      I=3
      N1=4
      J=3

      DO 10 K=1,N
        CALL INPUT(A)
        WRITE(20,101)K,((A(I,J),J=1,N),I=1,M)
        CALL TRIANG(IP,A,N)
        WRITE(20,102)K,((A(I,J),J=1,N),I=1,M)

10- 11      CONTINUE
12          STOP

      101    FORMAT(' BEFORE TRIANGULARIZATION ',I1,')'//3(3X,F20.10)
      102    FORMAT(' AFTER TRIANGULARIZATION ',I1,')'//3(3X,F20.10)

      END

      COMMON
      VARIABLES=1,
      ROUTINE=INPUT,
      A(4,3)=REAL(-2,1);

```

Figure 12. Listing of an Example Program (Page 1 of 2)

77

**Figure 12. Listing of an Example Program (Page 2 of 2)**

The first three cases of test run No. 1 (see Pages B-2 to B-4 in Appendix B) show coverage of code for the MAIN program as 100% in the column marked Summary. Subroutine TRIANG gets a summary coverage of 86.96%. The remaining segments to be tested are numbers 3, 12, and 16, as seen in the segment reference report (Page B-3 of Appendix B). The segment reference tables are used to relate the segment numbers and their corresponding program statement numbers together. As an example, it is seen that segment 3 contains lines 34, 35, 36, and 37 in subroutine TRIANG (see Figure 12). These lines correspond to:

```

        IF(A(K,K).EQ.0)34 IP(N) = 035
6      CONTINUE → K = K+136 IF (K.LE.N)37 loop
(DO-loop termination includes an implied conditional branch)

```

Following the summary reports and the segment reference tables, the trial statistics appear on Page B-4, for example. These are the  $X_i$  that are needed to calculate the estimate of the number of remaining tracks. (Actually, more than three cases are required for the estimation and the three entries on Page B-4 form only a part of the data used.)

Supplied as part of the testing package is a program that interacts with the user and calculates the difference of the two sides of the estimation equation in Paragraph 3.1.4 based on trial solutions supplied by the user.

To explain how the  $X_i$  are formed, the formation of  $X_1$  and  $X_2$  will be considered. Case 1 of run 1 (see Page B-2) shows the number of times each segment of MAIN and TRIANG were executed. Since this is the first test case, the first unique track is automatically formed. Case 2 of run 1 for TRIANG (Page B-2) shows the same segments being executed (the number of executions of each segment listed happen to be the same, but this is irrelevant, the comparison is made on the basis of whether or not the segment was executed, not on how many times) as in case 1, run 1. However, the MAIN routine shows a difference in execution. Therefore case 1 and case 2 are different, so we form  $X_1 = 1$ . This means that one case occurred since the last unique track. If we compare case 3 of run 1 against cases 1 and 2 we also find a difference in the MAIN routine (see segment 3 execution counts). This gives us our third unique track. Hence,  $X_2 = 1$ , also, since only 1 case occurred since the last unique track.

Continuing in this fashion, by comparing cases 1 through 9 (in Appendix B), in order, we find unique tracks for cases 1, 2, 3, 4, and 8. (A summary of the nine cases is found on Page B-10.) The Boolean tokens associated with the sequence are shown below:

CASE	1	2	3	4	5	6	7	8	9
NEW/ OLD (=1,=0)	1	1	1	1	0	0	0	1	0

$x_1=1$     $x_2=1$     $x_3=1$     $x_4=4$

By using the estimation equation, it was determined that there existed 9.1 new tracks to be found.

Appendix C contains reference tables of the APTS output for a constructed case. The constructed case shows the use of monitor variables (page C-2). For constructed cases, the user is required to supply input data to the program, and to supply the monitor variables. It is seen that the user-supplied input is in the DATA statement in the MAIN program. Subroutine TRIANG shows the use of monitors inserted into the program of branch points.

By analyzing the unexercised segments, 3, 12, and 16 of the three test runs of Appendix B, where they are marked by asterisks in all three of the segment reference tables (Pages B-3, B-6, B-9), it can be determined from the listing that the variable T holds the key to exercising these segments. Further examination suggests that if A[3,3] is equal to zero then segment 3 will be exercised.

Segment 12 requires variable T to be zero. For this to be true, A[1,1] could be equal to zero or |A[3,2]| could be greater than |A[2,2]| and A[1,2] must be equal to zero.

Segment 16 also requires variable T to be equal to zero. This condition will result if A[1,1] is not zero and A[1,2] is equal to zero.

These findings determined the initial values of A for the DATA statement. By observing the segment reference for subroutine TRIANG, we find that segments 3, 12, and 16 have been executed and the test coverage is complete.

### 3.3 ADDITIONAL PROBLEMS IN COVERAGE TESTING

#### 3.3.1 Formation of Execution Sequences

It is well to state at the outset that only the outline of this problem has been established. The following paragraphs describe the background and outline of the problem.

The use of tracks as proxies for execution sequences is in part necessary and in part expedient. Tracks are necessary because one usually cannot determine the actual sequence from a list of usages: with several entrances and several exits from a node and a different usage number of each, there is usually no way to determine the actual sequence of the computation that would produce the usage numbers. On the other hand, information often is available which would allow the program flow to be determined in a gross or general way, and that information heretofore has not been employed in our studies. It would be helpful to program testers to provide a general sequence of the flow resulting from a given input driver set.

To illustrate this, an example, depicted in Figure 13, shows the set of executed segments and their counts as solid lines or arcs between all nodes which were passed during the first data set employed on the ORLA program. It will be noted that dotted lines are also shown emanating from certain of the nodes which were passed. These are branches which were not taken on this run; they would be important in coverage testing but can be ignored for the present discussion. The flow of the computations can be determined unambiguously only in the cases where a single execution is performed on a segment and no other segment parallels the segment. For example, there is such a segment joining nodes 355 and 263 in the central lower one-third of the chart. This and others are highlighted in Figure 14, where they may be easier to locate.

The general flow can be formed from the unambiguous segments which show a usage of one. In one case, there are (at node 226) two segments, both with a count of 1, shown exiting a node. But this particular ambiguous case is easily resolvable (i.e., precedence determined) because the branch along segment 13, joining nodes 226 and 483, joins to the exit (END), and so cannot

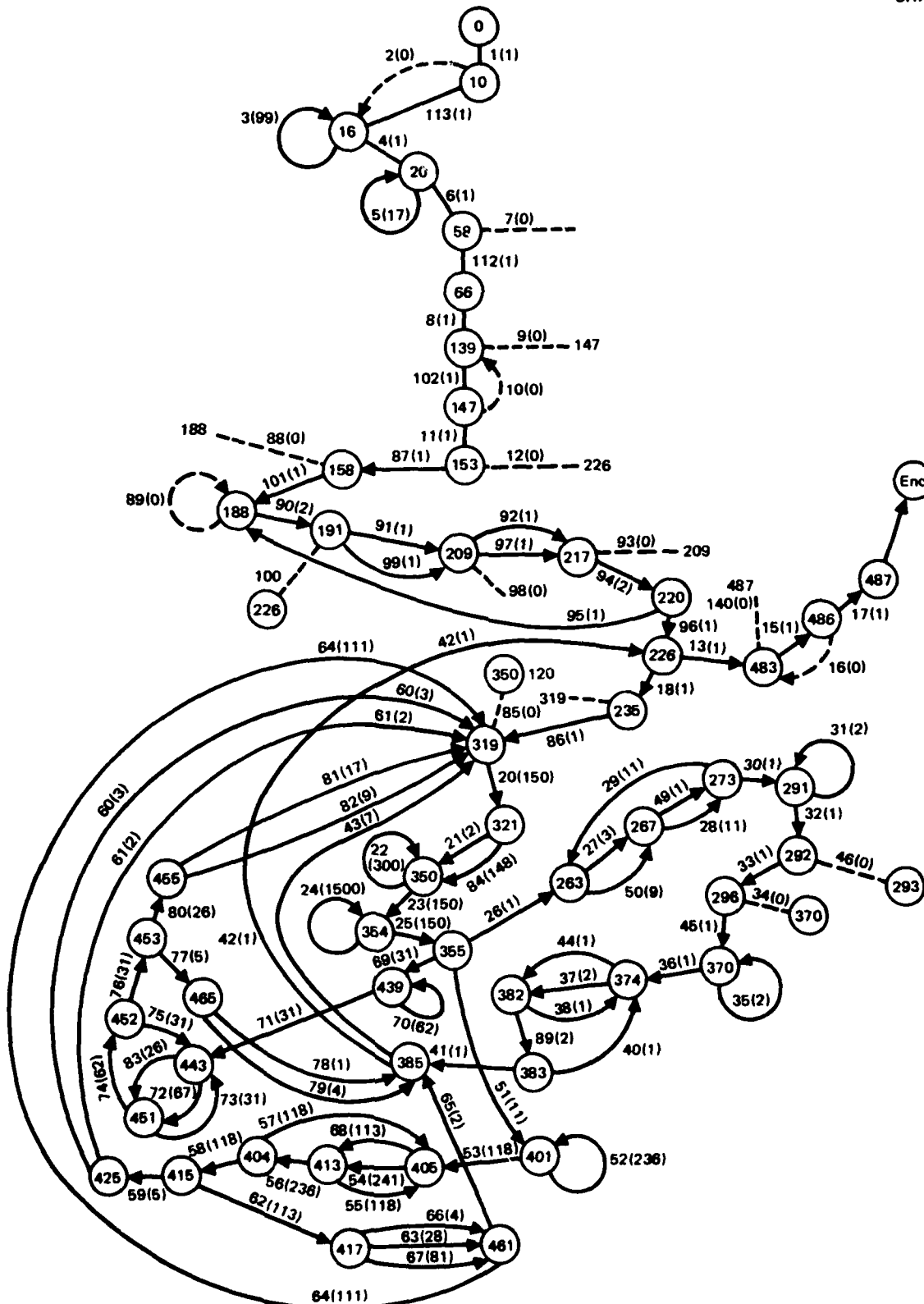


Figure 13. Response to First Data

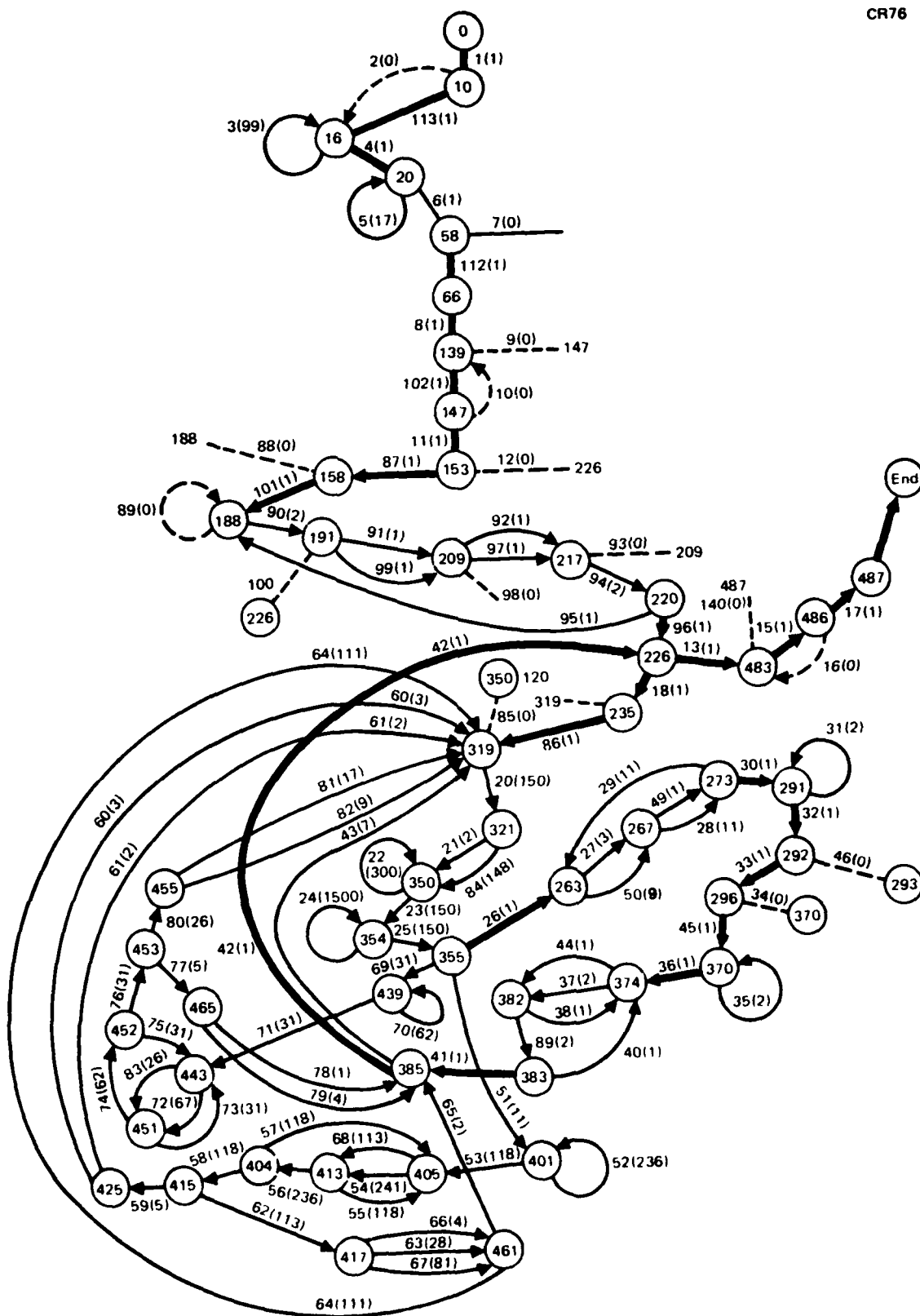


Figure 14. General Flow of Computation

precede the segment joining nodes 226 and 235. This suggests an interesting problem of which the preceding example is the most trivial: given a set of nodes and their counts, determine under what conditions the actual flow can be determined. This "academic" problem will not be pursued in this study.

The application of the simple rule which establishes the one-time and segments (a "footprint," or better, a "one-print") permits a linking of certain segments to form contiguous blocks of the program, the General Flow of the title of Figure 14.

Such linkings are shown in Figure 14, where the defined flow consists of the following:

- Block 1: Segments 1, 113, 4, 6, 112, 8, 102, 11, 87, 101
- Block 2: Segments 96, 18, 86
- Block 3: Segment 26
- Block 4: Segments 30, 32, 33, 45, 36
- Block 5: Segments 41, 42, 13, 15, 17 (END)

Even the undefined flow can be combined to form pseudo segments if there are not dotted lines: thus, the series/parallel segments 20, 21, 84, 22, 23, 24, and 25, which are between nodes 319 and 355, can be treated as a single pseudo segment with a usage of 150, the entry and exit counts at the two joined nodes. In addition to these pseudo segments, another type of merging is possible in certain areas. For example, some of the segments from Block 4 of the above list can be joined with the segment of Block 3 to form a super-block. Since all possible paths to and from nodes 263 and 273 have been exercised, these can be eliminated from further consideration, permitting formation of a pseudo segment with which to join segment 30 to segment 26. Also, since node 291 has all exits exercised, it too can join to form a larger block (26, pseudo segment, 30, and 32). Because node 292 has a dotted line out of it, there is no further merging possible between the two blocks.

Even though the remainder of the program flow is undefined, there are many points which are internal to the undefined blocks where reduction is possible. A trivial example is the pair of parallel paths 91 and 99 between nodes 191 and 209, which can be merged into a two-use segment; more interesting



cases can be identified in the lower left portion of Figure 13. Thus, between nodes 401 and 415 are segments 53, 68, 54, 55, 57, 56, and 58, all of which can be merged to a 118-use pseudo segment.

Figure 15 shows a considerably pruned version of the flow diagram. As with the preceding, it is developed from the one-prints and more is required to establish the sequence. For example, segment 42 appears to follow (dynamically) 41, but there is no reason to think, a priori, or in a local context that it actually does. In a global context, however, it is known that segment 42 is the later exit from node 385, because 42 joins to 226 and from there out to the END.

The primary purpose of investigation of the problem of pruning the flow diagram was to assist in the development of a display-aided test bed, where sections of the program could be showed in network form and successively pruned on a case by case basis.

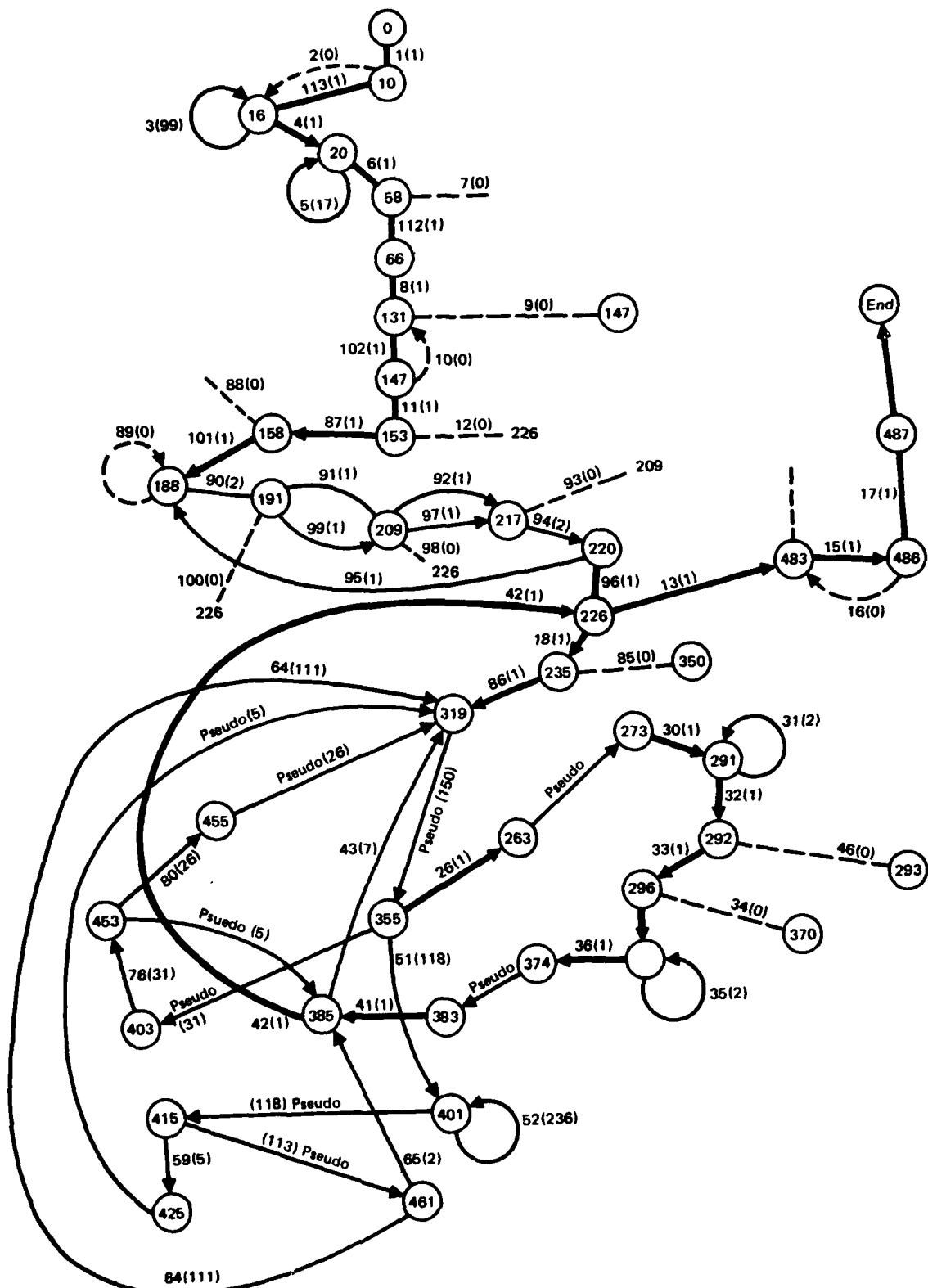
### 3.3.2 Partially Automated Test Bed

In keeping with the desire to achieve economical testing, the goal was to automate the entire process which has been outlined in the preceding discussion.

The major problems in completely automating the cover-testing process are in construction of the software required to establish the status of testing, maintain suspense files on all unexercised program segments, insert augmenting variables corresponding to predicates which define the entry into the (unexercised) computational segments, search the input variable space to achieve entry, compare the resulting track with previously obtained tracks, and prune the original tracks to a set of smaller dimension (manifested in the reduction of the original n-tuples to tuples of smaller size). This is to be done within the restricted physical environment of current displays and I/O equipment.

The complete list of tasks required is briefly summarized once again:

- A. Identify unexercised branches (at the end of the initial runs with random numbers).



**Figure 15. Partially Pruned Flow Diagram**

B. Pick an unexercised branch and display the listing associated with the branch (a "back" sort is required which identifies the instruction number of the involved predicate).

C. Formation of an auxiliary variable based on the nature of the predicate. (For example, if the test,  $A < B$ , is the predicate, the auxiliary variable could be  $C_1 = B - A$ ).

D. Create a variable (with requisite modifications to the object program). Recompile the program.

E. Vary input variables until the auxiliary variable is positive. Rationale for the variation depends on the program variables identified in the listing.

F. List all exercised segments and compare with preceding usage.

G. "Release" the variable and proceed to a new unexercised branch.

H. In an extension of the above procedure, several auxiliary variables can be inserted at one time and input data chosen in some systematic way (a search) to achieve arbitrary valuations on all auxiliary variables.

The results of a run or series of runs can be displayed in the form of a list of unexercised segments. It is clear that the information of the type shown in the bottom row of Table 1, can provide a quick look at the status of segment coverage after an initial set of runs has been made. The segment numbers marked by asterisk as, for example, on Page B-3 of Appendix B, can be displayed.

The back sort to identify the instruction number at the beginning of any chosen segment can be easily automated.

The process of inserting auxiliary variables at the predicates associated with unexercised segments at present must be done manually. The problem of inserting the variables requires a recompilation and this must be monitored. Development of the form or expression with which to represent the auxiliary variable may require scanning the listing over an extensive set of instructions.

## Section 4

### ERROR-DETECTION MODELS

#### 4.1 SUMMARY

Two variations of the Jelinski-Moranda model were developed for estimation during program development. The first permits estimation of the error content of the completed software package using data which is taken on only portions of the package. That model is applicable when the eventual size of the program is known at the outset.

The second model permits a similar analysis during the development of any software package which is homogeneous with respect to its complexity (error making and finding).

These models should assist analysts in an early determination of error content. They should also eliminate the present practice of applying models to the wrong regime (decreasing failure rate models applied to growing-in-size software).

#### 4.2 INTRODUCTION

In normal usage of the Jelinski-Moranda model, the software package under test is assumed to be of fixed size with a fixed number of incipient errors. The size of the package does not appear explicitly in the model as a parameter, and its effect is only indirectly realized by the way it affects the number of incipient errors which exist at the start of testing (there is a direct relation between instruction count and error count).

The model could not be employed legitimately on software packages which were incomplete. Several workers attempted to fit the model to an initial period of time when its error rate was, indeed, increasing, due to the growing size, and they met with no success. (As a matter of fact, the only models which produced reasonable estimates when applied during this regime, were the increasing failure rate models.)

It would be helpful if, at the outset, an estimate could be obtained of the total error count which will be realized in test and usage of a package.

Recent work by IBM (Reference 25) has prompted a reexamination of the original Jelinski-Moranda model for the purpose of incorporating the (changing) program size. This turns out to be very easily effected if good record keeping can be maintained during program development so that the size of the package is recorded as a function of some convenient timing metric (CPU or calendar). Following is a description of the analysis.

The original model is depicted in Figure 16, where the two parameters are shown in Figure 18(b), and a typical realization of the error-finding process is shown in Figure 16(a).  $N$  is the initial error content (of a completed program) and  $\phi$  is the contribution to the error rate due to a single error.

While the meaning of  $\phi$  is maintained in the two models, the meaning of "initial error content" needs to be clarified. This is done below in the description of Model 1, where, in effect,  $N$  maintains its meaning as the number of errors in a completed package. In the second model, a fixed error rate per instruction is assumed, and growth of the package is measured by the count of instructions (under test) versus time.

#### 4.2.1 Model 1

Let  $S(t)$  denote the fraction of the total number of statements which a complete program will have. The metric  $t$  is measured in terms either of the accumulated CPU time, or of the amount of calendar time, which has been used for testing the package.

The simplest way of introducing the effect is to use  $S(t)$  as a "modulation" of the error detection rate  $Z(t)$  of the original model. In the notation formerly employed, this combined or modulated rate, denoted  $W(t)$ , is:

$$\begin{aligned} W(t) &= S(t) Z(t) \\ &= S(t) [(N-i+1)\phi] \text{ for } T_{i-1}' \leq t \leq T_i' \end{aligned} \quad (2)$$

AD-A093 788

MCDONNELL DOUGLAS ASTRONAUTICS CO HUNTINGTON BEACH CA  
METRICS OF SOFTWARE QUALITY.(U)

F/G 9/2

NOV 80 Z JELINSKI, P MORANDA, J CHURCHWELL  
MDC-69326

F49620-77-C-0099

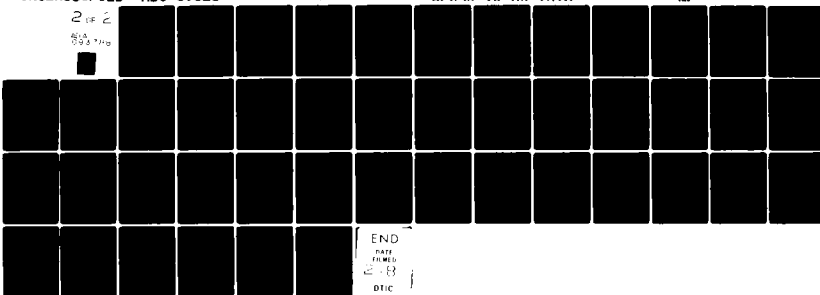
AFOSD-TR-80-1376

NI

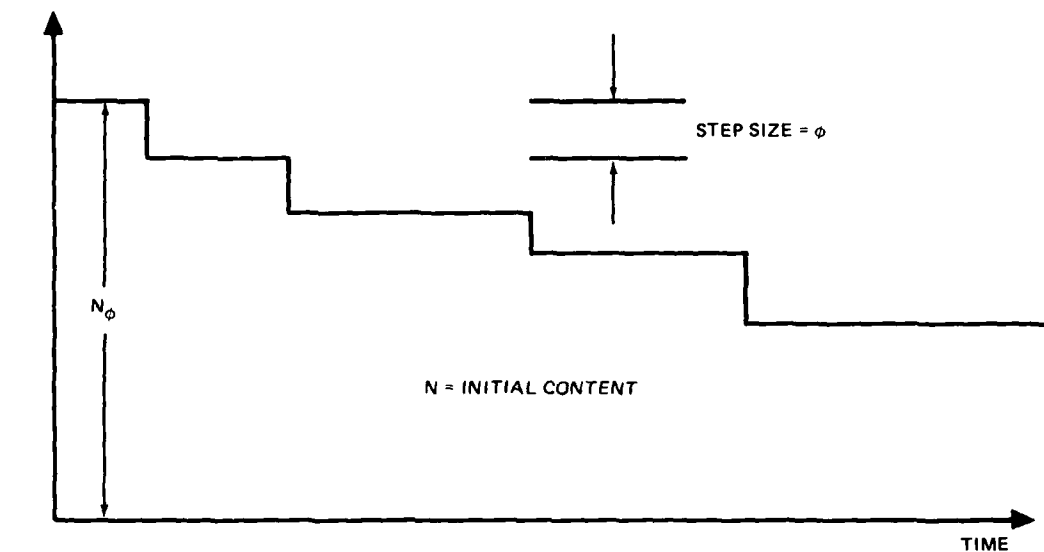
UNCLASSIFIED

2 OF 2

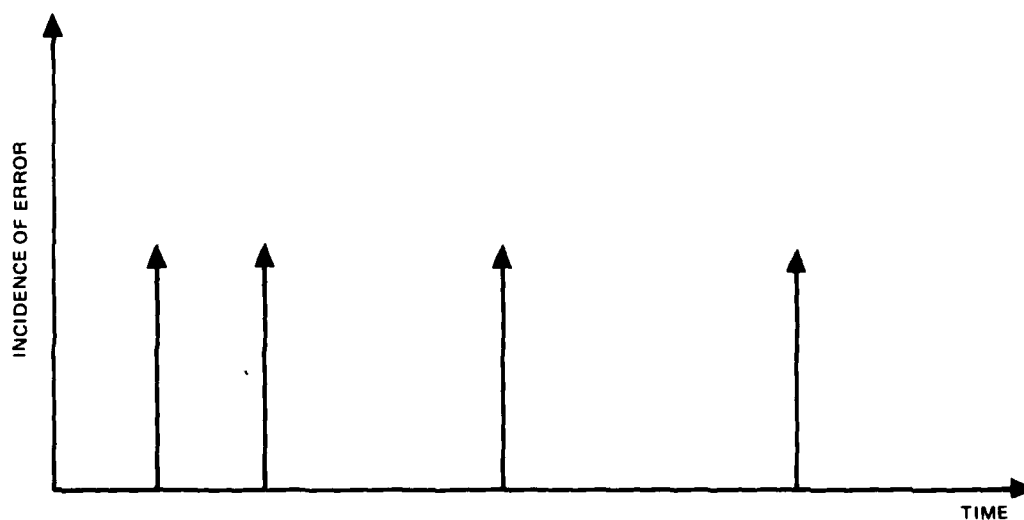
NOV 80 11:00



END  
PAGE  
FILMED  
G-1  
DTIC



(b) FAILURE RATE VERSUS TIME



(a) TYPICAL REALIZATION

Figure 16. Purification Process and its Realization

and  $T_1', T_2', T_3', \dots$ , denote the times of detection for the errors. (Primes are employed on  $T$ 's to distinguish them from the times of the original process.) The effect of  $S(t)$  on the  $T_i'$  should be made clear at the outset. When  $S(t)$ , the fraction of the total count, increases, the composite error rate will generally increase, as will the liability for error for the "modulated" process. For this reason, the times  $T_i'$  for the composite process,  $W(t)$ , will be different from the  $T_i$  of the  $Z(t)$  process. Since  $N$  in the original model represented the total error content of a complete software package, a proper correspondence which preserves the meaning is that  $N$  is the error content at a time corresponding to the completion of the software package,  $S(t) = 1.00$ . This necessarily presumes that the size of the package which will be developed is known at the outset. ( $S_0(t)$  would represent the fraction of the total which is accomplished at time  $t$ .) This may or may not be a serious barrier. Some modules can be sized at the outset, but large complex programs may not be. An alternative to this is offered subsequently in Model 2.

For the present model,  $S_0(t)$  is a nondecreasing function which starts at zero at  $T_0'$  and achieves its maximum value at some unknown-at-the-outset time,  $T_c'$ .

Thus,  $0 \leq S_0(t) \leq 1$ , with  $S_0(T_0') = 0$  and  $S_0(T_c') = 1$ .

While  $S_0(t)$  is, in the large sense, random, the records of progress will permit specific values of  $S_0(t)$  to be determined and the randomness is of no concern. In particular, it is necessary that  $S_0(t)$  can be determined at the epoch times  $T_1', T_2', \dots, T_n'$  at which the errors are detected.

When the completion time,  $T_c'$ , is reached and for times thereafter, the software package is complete ( $S_0(T_c') = 1$ ) and, formally, the density given in Equation (1) is the same as that given in the original paper (Reference 3).

It has been mentioned earlier that the time pattern of errors will be different for the "modulated" process, and it is interesting to see just what would happen if  $S_0(T_0')$ , or for short,  $S_0(0)$ , were 0.10 (10% of the package is initially available for test), and it did not increase beyond that for a long

period of testing. The time pattern of errors  $T_1^i, T_2^i, \dots, T_n^i$  which would occur, would have associated separations  $X_1^i = T_1^i - T_0^i, X_2^i = T_2^i - T_1^i, \dots, X_n^i = T_n^i - T_{n-1}^i$ .

Because  $S_0(0) = 0.10$ , the composite detection rate for the first error would be  $(0.10) N\phi$ , that is, 10% of the original error-detection rate. This means that the first detection time  $T_1^i$ , would (on the average) be 10 times as long as the time for the corresponding error of the unmodulated process. The second error would have the same property (on the average), and so forth. The implications of this fact can be seen from the following. The likelihood function would be

$$L(X_1^i, X_2^i, \dots, X_n^i) = \prod_{i=1}^n S_0(0) \phi [N-(i-1)] \exp \{-[S_0(0) \phi (N-i+1) X^i]\}.$$

The likelihood equations obtained by differentiating the logarithm of the likelihood with respect to  $N$  and  $\phi$  are:

$$\left. \begin{aligned} \sum_{i=1}^n \frac{1}{N-(i-1)} - S_0(0) \phi \sum_{i=1}^n X_1^i &= 0 \\ \text{and} \\ \frac{n}{\phi} - S_0(0) \sum_{i=1}^n [N-(i-1)] X_1^i &= 0 \end{aligned} \right\} \quad \begin{aligned} (3a) \\ \\ (3b) \end{aligned}$$

As noted above, the observables  $X_1^i$  would be (about or on average) 10 times as large as before. Thus, from Equation (3b), the solution  $\hat{\phi}$  will be (on average) the same as its value for the unmodulated process, or for the completed software package.

Using the solved-for value of  $\hat{\phi}$  in Equation (3a) and the fact that  $S_0(0)X_1^i$  in the new process is the same as  $X_1^i$  in the original process, it is seen that the solution  $N$  is also the same as before.

The analysis then shows that if it is known that a package under test represents (in all respects) a certain percentage of the total, then the total eventual error content can be estimated by using these slightly modified likelihood equations.

The result is encouraging for the outlook for success in the following simple generalization of the above example. In this generalization, the  $S_0(t)$  modulating function is constrained to be constant during each test interval. Using essentially the same notation as before, the likelihood equations for the generalized modulated process are

$$\sum_{i=1}^n \frac{1}{N-i+1} - \phi \sum_{i=1}^n S_{i-1} X_i^! = 0 \quad (4a)$$

and

$$\frac{n}{\phi} - \sum_{i=1}^n S_{i-1} (N-i+1) X_i^! = 0 \quad (4b)$$

where  $S_{i-1}$  is the percentage completion achieved prior to the start of the  $i^{\text{th}}$  interval.

Solutions for the parameters can be carried out as indicated above in the example.

The mean-time-to-next error MTTF ( $n+1$  st in the present context) can be estimated by evaluating the rate at time  $T_n^!$  and taking the reciprocal of it. In the present case (using a subscript on the left side to correspond to the model number):

$$\widehat{\text{MTTF}}_1 = \frac{1}{S(T_n^!) (\widehat{N}-n) \widehat{\phi}},$$

where  $\widehat{N}$  and  $\widehat{\phi}$  are solutions to the Maximum Likelihood Equations (MLE's).

#### 4.2.2 Model 2

Let  $E_p$  denote a characteristic rate of error-making for the programmer (or programmer team) and the program type. This rate will be estimated by application of the model described subsequently, but there are some useful facts concerning this parameter.

In 1975, it was observed (Reference 23) that there appears to be a "... 'universal' coding - error rate ...," which has a value of about 2 errors per 100 instructions (of the language in which the program has been written). This observations was based primarily on the data (now famous) provided by F. Akiyama, but also on earlier observations made by B. J. Hatter, et. al. Subsequently, the validity of this "thermodynamically stable" parameter has been reinforced by several other studies.

The interesting feature of some of this later data (Reference 24) is that the error rate of two per hundred was observed on programs which had completed their development and integration phases; they were under test before the relevant error counting was initiated. This is surprising since the coding error rate is thought of as being similar to a typist's miskeying, and should be purifiable by edit routines and by code checking due to early misstarts of the program.

These features of an hypothesized entity are fortunately not used in the following analysis.

The error rate at any point in the development of a program whose current instruction count is  $G(t)$  is assumed to be proportional to the current error content

$$V(t) = \phi[G(t) \cdot E_p - n(t)] \quad (5)$$

where  $n(t)$  is the accumulated number of error corrections, and  $E_p$  is the per instruction error rate.

As before, if  $G(t)$  can only change at error-discovery epochs,  $T_1, T_2, \dots, T_n$ , and, if  $n(t)$  also has this feature, then the rate has the form

$$V(t) = \phi[G_{i-1} E_p - (i-1)] \text{ for } T_{i-1} \leq t \leq T_i \quad (6)$$

where  $G_{i-1} = G(T_{i-1})$ , and  $n(t)$  is  $i-1$  for the interval starting at  $T_{i-1}$ .

Since  $G(t)$  is a function or process which takes place without any apparent dependence on the error-finding process (except that the error epochs are assumed to be the points of entry of new code) it is reasonable to assume that the random time separations between errors ( $X_1, X_2, \dots, X_n$ ) are statistically independent.

Under these conditions, the constant rate implies an exponential distribution for the  $X_i$ , and the likelihood function for  $n$  errors is:

$$L(X_1, X_2, \dots, X_n) = \sum_{i=1}^n \phi[G_{i-1} E_p - (i-1)] \exp \{-\phi X_i [G_{i-1} E_p - (i-1)]\} \quad (7)$$

The MLE's obtained by differentiating the logarithm of the likelihood function with respect to  $\phi$  and  $E_p$  are:

$$\sum_{i=1}^n \frac{G_{i-1}}{G_{i-1} E_p - (i-1)} - \phi \sum_{i=1}^n G_{i-1} X_i = 0 \quad (8a)$$

$$\frac{n}{\phi} - \sum [G_{i-1} E_p - (i-1)] X_i = 0 \quad (8b)$$

The MLE's are solved as before: Equation (8b) can be algebraically solved for  $\phi$ ; this is substituted in Equation (8a), and the resulting key equation is solved for  $E_p$  by trial and error.

It is recalled that the desired performance parameter is  $E_p$ , which can then be used with either the current (known) or projected (estimated) instruction count to determine the total error content.

Estimates of the MTTF at any time can be obtained by the formula

$$MTTF_2 = \frac{1}{\hat{\phi}[G_n \hat{E}_p - n]} \quad (9)$$

#### 4.3 CONCLUSIONS

The two models presented in the analysis are both very tractible analytically.

Model 1 would be of use for those programs whose eventual size is known at the outset. It requires that a record of the times of error occurrences be maintained as well as a record of the percentage of completion at each of the error-detection times. It provides, at any stage of testing, an estimate of the error content of the untested complete package.

Model 2 applies to any developing software package which is homogeneous with respect to the complexity of programming and with respect to the talents of the programmers. The important property is that  $E_p$ , the error-making rate (or error-finding rate), must be a constant across the entire software package. In case of inhomogeneity separate analyses are advised.

#### 4.4 GLOSSARY

Terms and symbols used in the preceding sections are identified as follows:

$S_0(0)$	A "modulation function" which ranges from $0 \leq S_0(t) \leq 1.0$ and is nondecreasing. It represents the fraction of the code completed up to time $t$ . It is a given for the problem.
$Z(t)$	The Jelinski-Moranda detection or purification process.
$W(t)$	The product of $S(t)$ and $Z(t)$ . It represents the error-making or error-detecting rate versus time for Model 1.
$N$	The number of errors in the completely coded software package. This is estimated from data.

$\phi$	The contribution of one error to the detection (failure) rate. This is estimated from data.
$T_i$	The time at which the $i^{\text{th}}$ error is found, measured in any convenient timing metric. An observable.
$X_i$	The separation between the $i^{\text{th}}$ and the $i-1^{\text{st}}$ error. An observable.
$n$	The cumulative number of errors found in testing up to time $T'_n$ .
$S_i$	Is the percentage of completion during the $(i+1)^{\text{st}}$ interval. This is provided as exogenous data.
$\hat{MTTF}_i$	The estimated meantime to error obtained by using Model $i$ ( $i = 1, 2$ ).
$G(t)$	The nondecreasing function representing the total instruction count of the package at time $t$ . This is a given for the problem.
$E_p$	The error making rate for a given program-programmer mix. It is estimated from data.
$n(t)$	The number of errors found during test up to time $t$ . An observable.
$V(t)$	A stochastic process representing error-making or error-detecting rate versus time.
$L(X_1, X_2, \dots, X_n)$	The generic representation for the likelihood function.

## REFERENCES

1. P. Moranda, "Quantitative Methods for Software Reliability Measurements", Final Report on AFOSR Contract F44620-74-C-0008, MDC Report G6553, December 1976.
2. L. G. Stucki, "Program Evaluation and Tester: PET", MDC Report M2085074, 1974.
3. W. Miller and D. L. Spooner, "Automatic Generation of Floating-Point Test Data", IEEE Transactions on Software Engineering, September 1976, Vol SE-2, No. 3.
4. D. J. Reifer, "A Glossary of Software Tools and Techniques", Computer, July 1977.
5. C. V. Ramamoorthy and S. F. Ho, "Testing Large Software with Automated Software Evaluation Systems", IEEE Transactions on Software Engineering March 1975; Vol. SE-1, No. 1.
6. J. Goodenough and S. L. Gerhart, "Toward a Theory of Test Data Selection", Proceedings of International Conference on Reliable Software, Los Angeles, California, 21-25 April 1975.
7. W. E. Howden, "Methodology for the Automatic Generation of Program Test Data", TR No. 41, McDonnell Douglas, February 1974.
8. B. Elspas, M. W. Green, K. N. Levitt, and R. J. Waldinger, "Research in Interactive Program Proving Techniques," SRI Report 8398-II, Standard Research Institute, 1972, Menlo Park, California.
9. J. King, "Symbolic Execution and Program Testing," Communications of the ACM, July 1976.
10. L. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," IEEE Transactions on Software Engineering, September 1976; SE-2, No. 3.
11. L. G. Stucki, "Program Evaluation and Tester: PET," McDonnell Douglas M2085074, 1974.
12. E. I. Cohen and L. J. White, "A Finite Domain-Testing Strategy for Computer Program Testing," (CSU-CISRC-TR-77-13), The Ohio State University, Columbus, Ohio, August 1977.
13. A. Fitzsimmons and T. Love, "A Review and Evaluation of Software Science", ACM Computing Surveys, Vol. 1, No. 1, March 1978.

14. P.B. Moranda, "Comments on A Review and Evaluation of Software Science", Surveyor's Forum, Computer Surveys, Vol 1, No. 3, September 1978.
15. J. L. Elshoff, "An Investigation into the Effects of the Counting Method Used on Software Science Measurements," IEEEETSE, Vol. SE-2, No. 4, December 1976.
16. T. J. McCabe, "A Complexity Measure." IEEE Transactions on Software Engineering, December 1976; Vol. SE-2, No. 4.
17. T. Gilb, Software Metrics, Winthrop Publishers, Inc., Cambridge, Mass. 1977.
18. B. Littlewood and J. L. Verrall, "A Bayesian Reliability Growth Model for Computer Software", Record, 1973 IEEE Symposium on Computer Software Reliability, New York, NY, 1973.
19. G. J. Schick and R. W. Wolverton, "An Analysis of Competing Software Reliability Models," IEEEETSE, March 1978; Vol. SE-4, No. 2.
20. K. Okumoto and A. Goel. "A Model for Reliability and Other Quantitative Measures of Software System Subject to Imperfect Debugging," Vol (of 5), RADC-TR-78-155, July 1979.
21. E. C. Miller. Tutorial on Program Testing Techniques. COMSAC77, Chicago, Illinois, 8-11 November 1977.
22. Z. Jelinski and P. B. Moranda. "Software Reliability Research" in Statistical Computer Performance Evaluation, Walter Freiberger, Ed.,
23. P. B. Moranda, "Estimation of A Priori Software Reliability," Computer Science and Statistics Interface Symposium, February 1975, Los Angeles, California.
24. T. A. Thayer, M. Lipow, and E. C. Nelson, "Software Reliability Study", Final Tech Report AD030798, TRW, Feb., 1976.
25. R. W. Motley and W. E. Brooks, "Statistical Prediction of Programming Errors", RADC-TR-77-175, Final Technical Report AD41106, Rome Air Development Center, AFSC, Griffiss Air Force Base, New York, May 1977.

Appendix A  
AUGMENTED ORLA PROGRAM



## PROGRAM ORLA

-- OPTIMUM REPAIR LEVEL ANALYSIS (ORLA)

-- O.R. JOHNSON ADCOM-WARNER RUBINS ALC-5508

-- THE CONSTANTS ARE ASSIGNED ACCORDINGLY-

1-BRT	2-CON	3-DLA	4-DLWP
5-DPL	6-DSS	7-FLA	8-FLWR
9-IAC	10-IPC	11-N	12-OSTICON
13-OSTIOS	14-OSTNCON	15-OSTNOS	16-PIUP
17-PSLRCON	18-PSLRPOS	19-PSMRCON	20-PSMRPOS
21-PWRCON	22-PWRPOS	23-RAC	24-RPC
25-SA	26-SRICON	27-SRPOS	28-SRNCON
29-SRPOS	30-TD	31-UF	32-UF
33-VD	34-VF	35-VI	36-ZI

-- THE VARIABLES ARE ASSIGNED ACCORDINGLY-

37-DF	38-DMR	39-FAP	40-FF
41-H	42-J	43-LA	44-LP
45-PP	46-ORA	47-RMB	48-SC
49-SW	50-UC	51-UW	52-W
53-X	54-YD	55-ZD	

-- THE COMPUTED VARIABLES ARE ASSIGNED ACCORDINGLY-

56-MTBD	57-UA	58-DAM	59-FA
60-FAM			

-- THE AGE COST VARIABLES ARE ASSIGNED ACCORDINGLY-

61-DCS	62-DUA	63-ICS	64-IUA
--------	--------	--------	--------

-- STORAGE TABLES

INTEGER      NVAR(18)      ,KF(11) ,KD(12) ,  
 -      KT(3) ,KSEN(6),JSEN(3),NUMK(3),  
 -      NUMJ(3),IPAS(100)

REAL      VAL(64)      ,FV(11) ,DV(12) ,  
 -      QCTGM(100)      ,TV(3) ,  
 -      HRS(100)      ,TTM(100) ,  
 -      AIA(100)      ,DAA(100)

REAL      ANSWER ,BLANK ,DASH ,NO ,  
 -      OUT(3) ,X      ,YES

DOUBLE PRECISION AGE(2,2)      ,AIR      ,C      ,  
 -      DATE(2)      ,DT      ,DY      ,  
 -      NAM(2,100,2)      ,NHA(3) ,NDH(4) ,  
 -      REA      ,PN(3)      ,PLS(2,3) ,  
 -      SVAR(10)      ,TC      ,VAR(64),  
 -      WUC(2)

-- LITERAL FORMATS OF DATA NAMES

LOGICAL      LB(10) ,LD      ,LA(3)

```

DATA      BLANK  ,DASH  ,NO    ,X    ,VFS
-          /1H    ,1H/   ,2HNO  ,1HX  ,3HYES /
DATA      AIR    ,REA    ,
-          /7HAIRLIFT ,7HNON-AIR /
DATA      VAR
-          /"BRT"  ,"CON"  ,"DLA"  ,"DLWR" ,"CPL"  ,
-          "DSS"  ,"FLA"  ,"FLWR" ,"IAC"  ,"IPC"  ,
-          "N"    ,"OSTICON" ,"OSTIOS" ,
-          "OSTNOCH" ,"OSTNOS" ,"PIUP" ,
-          "PSLRCON" ,"PSLRCS" ,
-          "PS4RCON" ,"PS4ROS" ,
-          "PWCON"  ,"PWPOS" ,
-          "PPC"    ,"SA"    ,"SPYCON" ,"SRICS" ,
-          "SRNCON" ,"SRNCS" ,"TS"    ,"UF"    ,
-          "UR"     ,"VD"    ,"VF"    ,"VI"    ,"ZI"    ,
-          "DF"     ,"DMR"   ,"FAQ"   ,"FF"    ,"U"     ,
-          "J"      ,"LA"    ,"LP"    ,"PP"    ,"QPA"   ,
-          "RM4"    ,"SC"    ,"SW"    ,"UC"    ,"UA"    ,
-          "N"      ,"X"     ,"VE"    ,"ZD"    ,"MTBD"  ,
-          "DA"     ,"DAM"   ,"FA"    ,"FAV"   ,"DCS"   ,
-          "DUA"    ,"TCS"   ,"TUA"   /

```

C

## -- RECORD FORMATS

C

C

2

FORMAT(8F9.2,8X)

3

FORMAT(1H1,10X,"CONSTANTS USED IN THIS RUN",5X,"DATE",2X,  
A8,2X,A8//)

4

FORMAT(2X,A7,F12.4,4X,A7,F12.4,4X,A7,F12.4)

5

FORMAT(I3,5X,7A9)

6

FORMAT(F8.0,4F5.2,2X,I1)

8

FORMAT(8F9.2)

809

FORMAT(4F9.2,4X,3I2)

119

FORMAT(20X,"ECONOMIC/SENSITIVITY ANALYSIS",6X,"NUMBER",I3//)

9

FORMAT(2X,I4,4X,2A8,A1,1Y,3A8,A2,2A8,A2)

109

FORMAT(/11X,"K FACTORS: K1",F5.2," K2",F5.2," K3",F5.2,  
" K4",F5.2)

10

FORMAT(/6X,"DESIGN MTRF",F8.0,2X,"MTBD",F10.1," UC",  
F8.0,3X,A7)

11

FORMAT(I1,9X,4A10,F12.2,F5.2,I3)

12

FORMAT(I3,7X,3A10,F3.1)

13

FORMAT(1H1,7X,"INTERMEDIATE MULTIPLE SUPPORT AGE SUMMARY",  
8X,"DATE",2X,A8,2X,A8//)

14

FORMAT(" AGE NOMENCLATURE",11X,"WUC",4X,"AGE COST",3X,  
"HRS AVAIL",2X,"AGE SETS NEEDED//)

15

FORMAT(1H,3A8,A2,1X,A5,F9.2,2X,F8.2,6X,I5)

16

FORMAT(/1X,"ITEM NO. DEMANDS/ MTTT REQUIRED",  
" %TOTAL SHARE OF AGE COST "/13X,"MONTH",11X,  
"HOURS",5X,"HOURS",5X,"COST",5X,"ALLOCATION//)

17

FORMAT(2X,I4,4X,F7.1,4X,F5.1,F9.1,F9.2,2(2X,F10.2))

18

FORMAT(/9X,"TOTALS",10X,F10.1,F9.2,2F12.2)

19

FORMAT(1H1,11X,"DEPOT MULTIPLE SUPPORT AGE SUMMARY",4X,  
"DATE",2X,A8,A2//)

21

FORMAT(/26X,"ECONOMIC ANALYSIS//)

```

22      FORMAT(/26X,"DEPOT",6X,"INTERMEDIATE",6X,"DISCARD"/)
550     FORMAT(/26X,"INPUT DATA VALUES"/)
23      FORMAT(1X,"BASE STOCK LEVEL",6X,I9,6X,I9,6X,I9)
551     FORMAT(4X,A7,F12.4,2X,A7,F12.4,2X,A7,F12.4)
24      FORMAT(" AGE",19X,I9,6X,I9)
25      FORMAT(" AGE MAINT.",12X,I9,6X,I9)
129     FORMAT(" ITEM NR",2X,"PART NUMBER",7X,"NOMENCLATURE",14X,
-       "NEXT HIGHER ASSEM"/)
26      FORMAT(" TECH DATA",13X,I9,6X,I9)
27      FORMAT(" TRAINING",14X,I9,6X,I9)
28      FORMAT(" PACKING / SHIPPING",4X,I9,6X,I9,6X,I9)
29      FORMAT(" SAFETY STOCK",10X,I9)
30      FORMAT(" LABOR",17X,I9,6X,I9)
31      FORMAT(" SPECIAL FACILITIES",4X,I9,6X,I9)
32      FORMAT(" REPAIR MATERIAL",7X,I9,6X,I9)
552     FORMAT(4X,A7,F12.4,2X,A7,F12.4/)
149     FORMAT(/)
33      FORMAT(" ITEM ENTRY",12X,I9,6X,I9)
34      FORMAT(" SUPPLY ADMIN.",24X,I9)
35      FORMAT(" PIPELINE SPARES",7X,I9)
36      FORMAT(" REPLACEMENT SPARES",34X,I9)
37      FORMAT(/5X,"TOTAL",12X,I10,5X,I10,5X,I10)
40      FORMAT(2X,"LIMITS- FROM 20% TO 500% OF ORIGINAL FACTOR",
-       " VALUE WITHIN 1% /10X,"PRINTED AT REVERSAL"/)
41      FORMAT(I4,3A8,A3,F8.0,F8.0,3A1)
411     FORMAT(2X,I3,4X,3A8,A1,F8.0,F9.0,2(5X,A1),6X,A1)
42      FORMAT(1H1,25X,"SENSITIVITY ANALYSIS"/)
43      FORMAT(/7X,"FACTOR",6X," % ORIG ",6X,"VALUE",6X,"DEPOT",
-       6X,"INTER",4X,"DISCARD")
44      FORMAT(/7X,A7,14X,F9.2,1X,3I12,4X)
45      FORMAT(/18X,F6.0,4X,F9.2,1X,3I12,4X/)
412     FORMAT(I8,2A8,A2,3A8,A3,2A8,A2)
46      FORMAT(/21X,"REPAIR LEVEL SUMMARY",15X,"DATE",2X,A9,2X,A3/)
47      FORMAT(3X,"ITEM",30X,"UNIT",14X,"REPAIR LEVEL",5X)
48      FORMAT(2X,"NUMBER",3X,"NOMENCLATURE",13X,"PRICE",5X,
-       "MTBD",3X,"DEPOT INTER DISCARD"/)
49      FORMAT(/)
556     FORMAT(7X,"NOMENCLATURE",5X,"NPA",10X,"DATE",5X,A8)
6010    FORMAT(/ DO YOU WISH AN EXPLANATION OF THIS PROGRAM?",
-       "(YES/NO)",5)
6011    FORMAT(A4)
6012    FORMAT(1X,A4)
6020    FORMAT(/20X,"ORLA      COST      MODEL"/)
6030    FORMAT(" ENTER CONSTANT VALUES (36 VALUES) IN ORDER AS LISTED")
9350    FORMAT(7X,10(A7,5X))
991     FORMAT(1X,10F12.4)
9353    FORMAT(/7X,9(I3,9X))
6040    FORMAT(/)
6050    FORMAT(/ ANY ADDITIONAL CONSTANTS/VARIABLES FOR SENSITIVITY ",
-       "ANALYSIS?")
6055    FORMAT(/ HOW MANY? (LIMIT 10)")
9992    FORMAT(1X,I2)

```

```

1620  FORMAT(/" NAME ",I3," ADDITIONAL CONSTANTS/VARIABLES",
-      " (USE",I3," LINES)")
1621  FORMAT(A8)
1622  FORMAT(1X,10A8)
6060  FORMAT(" INCORRECT NAME")
1640  FORMAT(2X,A8," DROPPED FROM ANALYSIS.")
6065  FORMAT(/" ENTER THE NUMBER OF ITEMS TO BE RUN IN THIS ANALYSIS")
6070  FORMAT(18X,"ENTER ITEM DATA")
1710  FORMAT(1X,"ENTER PART NUMBER, NOMENCLATURE, NEXT HIGHED ",
-      "ASSEMBLY"/10X,"FOR ITEM NUMBER ",I3,4X,"(USE 3 LINES)")
1711  FORMAT(3A8/4A8/3A8)
1712  FORMAT(1X,3A8/1X,4A8/1X,3A8)
6080  FORMAT(" ENTER MEAN TIME BETWEEN FAILURE, Y FACTORS (4 VALUES)")
6090  FORMAT(" AND SHIPPING CODE (0 = AIRLIFT, 1 = NON-AIRLIFT) ")
6091  FORMAT(1X,F8.0,4F6.2,I4)
6100  FORMAT(" ENTER VARIABLES (23 VALUES) FOR THIS ITEM IN THE ",
-      "PROPER ORDER")
9354  FORMAT(/7X,10(I2,10X))
2955  FORMAT(/" ANY CORRECTIONS? (YES/NO)")
9352  FORMAT(7X,10(A7,5X))
6110  FORMAT(" DO YOU WISH TO RUN AN AGE SUMMARY COMPUTATION?",
-      " (YES/NO)")
2000  FORMAT(/" DO YOU WANT AN EXPLANATION OF AGE SUMMARY? (YES/NO)")
6120  FORMAT(/" ENTER THE NUMBER OF AGE SUMMARIES TO BE RUN")
6140  FORMAT(/" ENTER TYPE OF AGE,COST, AVAILABILITY/HOURS, AND"/
-      " THE NUMBER OF ITEMS YOU HAVE FOR THE AGE SUMMARY")
6130  FORMAT(" ENTER AGE NOMENCLATURE (MAXIMUM OF 26 CHAR)"/
-      " AND JUC UNIT CODE (USE 2 LINES)")
6151  FORMAT(3A8,A2/2A8)
6152  FORMAT(1X,3A8,A2/1X,2A8)
6153  FORMAT(1X,I1,3F10.2)
6160  FORMAT(/" ENTER THE ITEM NUMBER AND MEAN TIME TO TEST")
6161  FORMAT(1X,I4,F10.2)
2997  FORMAT(F9.0,4F6.2,I1,7F10.2)
2998  FORMAT(5F10.2)
2999  FORMAT(11F10.2)

```

C

C

C

-- OPEN LOGICAL DEVICES

0- 1

2

3

4

OPEN(UNIT=6,DEVICE="DSK:")

OPEN(UNIT=4,DEVICE="DSK:",DIALOG="ORLA.INP",ACCESS="SEQIN")

OPEN(UNIT=10,DEVICE="DSK:",ACCESS="SEQINOUT",DISPOSE="DELFTE")

OPEN(UNIT=12,DEVICE="DSK:",ACCESS="SEQINOUT",DISPOSE="DELFTE")

C

C

C

-- ADD CODE TO ALLOW RECOVERY OF STATISTICS UPON ABNORMAL EXIT

5

6

7

8

ASSIGN 510 TO IREEN

CALL REEN(IREEN)

IOUTPT=6

WRITE(IOUTPT,6010)

C

9

10

INPUT=4

IWORK1=10

```

11      IWORK2=12
      C
      C      -- REPLACE WITH RANDOM YES/NO
      C
      C      READ(INPUT,6011)ANSWER
12      CALL ASK(ANSWER)
      C      -- WRITE(IOUTPT,6012)ANSWER
13- 14    IF(ANSWER.EQ.YES)CALL XPLAIN(IOUTPT,VAR)
      C
      C      -- INITIAL VARIABLES, TABLE
      C
15      ICOUNT=0
16      DO 51 N=1,100
17          AIA(N)=0.0
18          DAA(N)=0.0
19- 20    51    CONTINUE
21      DO 511 I=1,18
22          NVAR(I)=0
23- 24    511   CONTINUE
25      NVAR(1)=31
26      NVAR(2)=32
27      NVAR(3)=47
28      NVAR(4)=50
29      NVAR(5)=56
30      NVAR(6)=62
31      NVAR(7)=64
      C
      C      -- ?
      C
32      WRITE(IOUTPT,6020)
      C
      C      -- READ CONSTANT VALUES
      C
33      CALL DATEV(DATE(1),DATE(2))
34      WRITE(IOUTPT,6030)
35      WRITE(IOUTPT,9353)(I,I=1,9)
36      WRITE(IOUTPT,9350)(VAR(I),I=1,9)
      C
      C
      C
37      INPUT=4
      C
      C      -- REPLACE WITH RANDOM REAL
      C
      C      READ(INPUT,*)(VAL(I),I=1,9)
38      CALL REAL4(VAL(1),9,-2,2)
39      WRITE(IOUTPT,9991)(VAL(I),I=1,9)
40      WRITE(IOUTPT,9353)(I,I=10,18)
41      WRITE(IOUTPT,9350)(VAR(I),I=10,18)
      C
      C      -- REPLACE WITH RANDOM REAL
      C
      C      READ(INPUT,*)(VAL(I),I=10,18)

```

```

42      CALL REAL4(VAL(10),9,-2,2)
43      WRITE(IOUTPT,9991)(VAL(I),I=10,18)
44      WRITE(IOUTPT,9353)(I,I=19,27)
45      WRITE(IOUTPT,9350)(VAR(I),I=19,27)

      C
      C
      C
      C
      -- REPLACE WITH RANDOM REAL

      READ(INPUT,*)(VAL(I),I=19,27)
46      CALL REAL4(VAL(19),9,-2,2)
47      WRITE(IOUTPT,9991)(VAL(I),I=19,27)
48      WRITE(IOUTPT,9353)(I,I=28,36)
49      WRITE(IOUTPT,9350)(VAR(I),I=28,36)

      C
      C
      C
      C
      -- REPLACE WITH RANDOM REAL

      READ(INPUT,*)(VAL(I),I=28,36)
50      CALL REAL4(VAL(28),9,-2,2)
51      WRITE(IOUTPT,9991)(VAL(I),I=28,36)
52      WRITE(IOUTPT,2955)
53      INPUT=4

      C
      C
      C
      C
      -- REPLACE WITH RANDOM YES/NO

      READ(INPUT,6011)ANSWER
54      CALL ASK(ANSWER)
55- 56      IF(ANSWER.EQ.YES)CALL CORECT(INPUT,IOUTPT,VAL,ANSWER,VAR)
57      WRITE(IOUTPT,6050)
58      INPUT=4

      C
      C
      C
      C
      -- REPLACE WITH RANDOM YES/NO

      READ(INPUT,6011)ANSWER
59      CALL ASK(ANSWER)
      C
      WRITE(IOUTPT,6012)ANSWER
60- 61      IF(ANSWER.EQ.NO)GOTO 50
62      WRITE(IOUTPT,6055)
      C
      C
      C
      C
      -- REPLACE WITH RANDOM INTEGER

      READ(INPUT,*)IT
63      CALL INT4(IT,1,1,10)
      C
      WRITE(IOUTPT,9992)IT
64- 65      IF(IT.GT.10)GOTO 1600
66      WRITE(IOUTPT,1620)IT,IT

      C
      C
      C
      C
      -- REPLACE WITH RANDOM INDEX OF VAR

      READ(INPUT,1621)(SVAR(J),J=1,IT)
67      CALL CHRVAP(VAR,SVAR,IT)
      C
      WRITE(IOUTPT,1622)(SVAR(J),J=1,IT)
68      IJ=0
69      DO 1630 I=1,IT
70      DO 1635 J=1,64

```

```

71- 72                                IF(SVAR(I).EQ.VAR(J))GOTO 1638
73- 74    1635    CONTINUE
75                                WRITE(IOUTPT,6060)
76                                IJ=IJ+1
77                                WRITE(IOUTPT,1640)SVAR(I)
78                                GOTO 1630
79                                NVAR(7+I-IJ)=J
80- 81    1638    CONTINUE
          1630
          C
          C    -- COMPUTE QCTGN FOR EACH PASS
          C
82                                WRITE(IOUTPT,6040)
83    50    CONTINUE
84                                WRITE(IOUTPT,6065)
          C
          C    -- REPLACE WITH RANDOM INTEGER
          C
          C    READ(INPUT,*)IT
85                                CALL INT4(IT,1,1,10)
          C    WRITE(IOUTPT,9992)IT
86                                DO 5550 K=1,IT
87                                    WRITE(IOUTPT,6070)
88                                    WRITE(IOUTPT,149)
89                                    WRITE(IOUTPT,1710)K
          C
          C    -- REPLACE WITH RANDOM CHARACTER
          C
          C    READ(INPUT,1711)PN,NOM,NHA
90                                CALL CHAR8(PN,3)
91                                CALL CHAR8(NOM,4)
92                                CALL CHAR8(NHA,3)
          C    -- WRITE(IOUTPT,1712)PN,NOM,NHA
93                                IPASS=K
94                                WRITE(WORK1,412)IPASS,PN,NOM,NHA
95                                WRITE(IOUTPT,6040)
96                                WRITE(IOUTPT,6080)
97                                WRITE(IOUTPT,6090)
          C
          C    -- REPLACE WITH RANDOM REAL/INTEGER
          C
          C    READ(INPUT,*)XMTBF,XK1,XK2,XK3,XK4,LIFT
98                                CALL REAL4(XMTBF,1,-2,2)
99                                CALL REAL4(XK1,1,-2,2)
100                               CALL REAL4(XK2,1,-2,2)
101                               CALL REAL4(XK3,1,-2,2)
102                               CALL REAL4(XK4,1,-2,2)
103                               CALL INT4(LIFT,1,0,1)
104                               WRITE(IOUTPT,6091)XMTBF,XK1,XK2,XK3,XK4,LIFT
105                               WRITE(IOUTPT,6040)
106                               WRITE(IOUTPT,6100)
107                               WRITE(IOUTPT,9354)(I,I=1,10)
108                               WRITE(IOUTPT,9752)(VAR(I),I=37,46)
109                               INPUT=1

```

```

C
C
C
C
110      -- REPLACE WITH RANDOM REAL
111      READ(INPUT,*)(VAL(I),I=37,46)
112      CALL REAL4(VAL(37),10,-2,2)
113      WRITE(IOUTPT,9991)(VAL(I),I=37,46)
114      WRITE(IOUTPT,9354)(I,I=11,19)
115      WRITE(IOUTPT,9352)(VAR(I),I=47,55)
116
C
C
C
C
114      -- REPLACE WITH RANDOM REAL
115      READ(INPUT,*)(VAL(I),I=47,55)
116      CALL REAL4(VAL(47),9,-2,2)
117      WRITE(IOUTPT,9991)(VAL(I),I=47,55)
118      WRITE(IOUTPT,9354)(I,I=20,23)
119      WRITE(IOUTPT,9352)(VAR(I),I=61,64)
120
C
C
C
C
118      -- REPLACE WITH RANDOM REAL
119      READ(INPUT,*)(VAL(I),I=61,64)
120      CALL REAL4(VAL(61),4,-2,2)
121      WRITE(IOUTPT,9991)(VAL(I),I=61,64)
122      WRITE(IOUTPT,2955)
123
C
C
C
C
121      -- REPLACE WITH RANDOM YES/NO
122      READ(INPUT,6011)ANSWER
123      CALL ASK(ANSWER)
124      -- WRITE(IOUTPT,6012)ANSWER
125      IF(ANSWER.EQ.YES)CALL CORRECT(INPUT,IOUTPT,VAL,ANSWER,
126      VAR)
127      WRITE(IWORK1,2997)XMTBF,XK1,XK2,XK3,XK4,LIFT,
128      (VAL(I),I=37,43)
129      WRITE(IWORK1,2998)(VAL(I),I=44,49)
130      WRITE(IWORK1,2999)(VAL(I),I=49,55),(VAL(J),J=61,64)
131      VAL(56)=XMTBF/(XK1*XK2*XK3*XK4)
132      QCTGM(IPASS)=VAL(31)*VAL(32)*VAL(46)/VAL(56)
133      5550 CONTINUE
134
C
C
C
C
131      -- AGE SUMMARY COMPUTATION
132      WRITE(IOUTPT,6040)
133      WRITE(IOUTPT,6110)
134      INPUT=4
135
C
C
C
C
134      -- REPLACE WITH RANDOM YES/NO
135      READ(INPUT,6011)ANSWER
136      CALL ASK(ANSWER)
137      -- WRITE(IOUTPT,6012)ANSWER
138      IF(ANSWER.EQ.NO)GOTO 65
139
C
C
C
C
135      -- AGE COMPUTATION ROUTINE

```

```

C      -- ITAGE=1 INTERMEDIATE , =2 DIRECT
C
137    WRITE(IOUTPT,2000)
C
C      -- REPLACE WITH RANDOM YES/NO
C
C      READ(INPUT,6011)ANSWER
138    CALL ASK(ANSWER)
C      -- WRITE(IOUTPT,6012)ANSWER
139- 140  IF(ANSWER.EQ.YES)CALL AGFTLK(IOUTPT)
C
141    WRITE(IOUTPT,6120)
C
C      -- REPLACE WITH RANDOM INTEGER
C
C      READ(INPUT,*)IT
142    CALL INT4(IT,1,1,10)
C      -- WRITE(IOUTPT,9992)IT
143    DO 553 L=1,IT
144        WRITE(IOUTPT,6040)
145        WRITE(IOUTPT,6130)
C
C      -- REPLACE WITH RANDOM CHARACTER
C
C      READ(INPUT,6151)((AGE(I,J),J=1,2),I=1,2),WUC
146    CALL CHAR8(AGE,4)
147    CALL CHAR8(WUC,2)
148    WRITE(IOUTPT,6152)((AGE(I,J),J=1,2),I=1,2),WUC
149    WRITE(IOUTPT,6140)
C
C      -- REPLACE WITH RANDOM INTEGER/REAL
C
C      READ(INPUT,*)ITAGE,AC,AH,RNAGE
150    CALL INT4(ITAGE,1,0,9)
151    CALL REAL4(AC,1,-2,2)
152    CALL RFAL4(AH,1,-2,2)
153    CALL REAL4(RNAGE,1,-2,2)
154    WRITE(IOUTPT,6153)ITAGE,AC,AH,RNAGE
C      -- WRITE(IOUTPT,6150)
155    NAGE=RNAGE
156    THRS=0.0
157    DO 57 I=1,NAGE
158        WRITE(IOUTPT,6160)
C
C      -- REPLACE WITH RANDOM INTEGER/REAL
C
C      READ(INPUT,*)IPAS(I),TTTM(I)
159    CALL INT4(IPAS(I),1,1,10)
160    CALL REAL4(TTTM(I),1,-2,2)
161    WRITE(IOUTPT,6161)IPAS(I),TTTM(I)
162    IP=IPAS(I)
163    HRS(I)=QCIGM(IP)*TTTM(I)
164    THPS=THPS+HPS(I)

```

```

165- 166    57    CONTINUE
167          XNAS=AIN(THRS/AH+.99999)
168          NAS=XNAS
169          IF(ITAGE-2)54,55,65
170          54    WRITE(IOUTPT,49)
171          WRITE(IOUTPT,13)DATE(1),DATE(2)
172          GOTO 56
173          55    WRITE(IOUTPT,49)
174          WRITE(IOUTPT,19)DATE(1),DATE(2)
175          56    WRITE(IOUTPT,14)
176          WRITE(IOUTPT,15)((AGE(I,J),J=1,2),I=1,2),
                    WUC(1),AC,AH,NAS
177          WRITE(IOUTPT,16)
178          TSHAR=0.0
179          TFRAC=0.0
180          TXTA=0.0
181          DO 63 I=1,NAGE
182              IP=IPAS(I)
183              FRAC=HRS(I)/THPS
184              SHARE=FRAC*AC
185              XIA=SHARE*XNAS
186              FRAC=FRAC*100.0
187              IF(ITAGE-2)60,61,65
188              60    AIA(IP)=XIA
189              GOTO 62
190              61    DAA(IP)=XIA
191              62    WRITE(IOUTPT,17)IPAS(I),QCTGM(IP),TTTT(I),
                    HRS(I),FRAC,SHARE,XIA
192              TFRAC=TFRAC+FRAC
193              TSHAR=TSHAR+SHARE
194              TXIA=TXIA+XIA
195- 196    63    CONTINUE
197          WRITE(IOUTPT,13)THRS,TFRAC,TSHAR,TXIA
198- 199    553    CONTINUE
199          C
200          C    -- ORLA PASS ROUTINE
201          C
202          65    REWIND IWORK1
203          C
204          C    -- WRITE CONSTANTS FOR RUN
205          C
206          69    WRITE(IOUTPT,49)
207          WRITE(IOUTPT,3)DATE(1),DATE(2)
208          WRITE(IOUTPT,4)(VAL(I),VAL(I),I=1,36)
209          70    READ(IWORK1,412,END=500)IPASS,PN,NUM,MHA
210          71    READ(IWORK1,2997)XMTBF,XK1,XK2,XK3,XK4,LIFT,(VAL(I),I=37,43)
211          READ(IWORK1,2998)(VAL(I),I=44,48)
212          READ(IWORK1,2999)(VAL(I),I=49,55),(VAL(J),J=61,64)
213          VAL(56)=XMTBF/(XK1*XK2*XK3*XK4)
214          IF(LIFT)73,72,73
215          72    OSTC=VAL(12)
216          OSTO=VAL(13)
217          SR=VAL(26)

```

```

213      SRI=VAL(27)
214      TC=AIR
215      GOTO 74
216      73      DSTC=VAL(14)
217      DSTO=VAL(15)
218      SR=VAL(28)
219      SRI=VAL(29)
220      TC=REA
221      74      WRITE(IOUTPT,49)
222      WRITE(IOUTPT,119)IPASS
223      WRITE(IOUTPT,129)
224      WRITE(IOUTPT,9)IPASS,PN,NOM,NHA
225      WRITE(IOUTPT,10)XMTBF,VAL(56),VAL(50),TC
226      WRITE(IOUTPT,109)XK1,XK2,XK3,XK4
227      ASSIGN 75 TO JUMP
228      GOTO 100

      C
      C      -- PRINT ROUTINE FOR ECONOMIC ANALYSIS
      C
229      75      WRITE(IOUTPT,550)
230      WRITE(IOUTPT,551)(VAR(I),VAL(I),I=37,54)
231      WRITE(IOUTPT,551)VAR(55),VAL(55),(VAR(I),VAL(I),I=61,62)
232      WRITE(IOUTPT,552)(VAR(I),VAL(I),I=63,64)
233      WRITE(IOUTPT,49)
234      WRITE(IOUTPT,21)
235      WRITE(IOUTPT,22)
236      DO 80 I=1,12
237      IF(I-3)76,76,77
238      76      KD(I)=DV(I)
239      KF(I)=FV(I)
240      KT(I)=TV(I)
241      77      IF(I-11)78,78,79
242      78      KD(I)=DV(I)
243      KF(I)=FV(I)
244      GOTO 80
245      79      KD(I)=DV(I)
246- 247      80      CONTINUE
248      WRITE(IOUTPT,23)KD(8),KF(5),KT(3)
249      WRITE(IOUTPT,24)KD(2),KF(1)
250      WRITE(IOUTPT,25)KD(3),KF(2)
251      WRITE(IOUTPT,26)KD(4),KF(3)
252      WRITE(IOUTPT,27)KD(5),KF(4)
253      WRITE(IOUTPT,28)KD(6),KF(7),KT(2)
254      WRITE(IOUTPT,29)KD(7)
255      WRITE(IOUTPT,30)KD(9),KF(6)
256      WRITE(IOUTPT,31)KD(10),KF(3)
257      WRITE(IOUTPT,32)KD(11),KF(10)
258      WRITE(IOUTPT,33)KD(12),KF(11)
259      WRITE(IOUTPT,34)KF(8)
260      WRITE(IOUTPT,35)KD(1)
261      WRITE(IOUTPT,36)KT(1)
262      WRITE(IOUTPT,37)KDT,KFT,KTT

```

C

```

C      -- WRITE TO WORK TAPE THE REPAIR LEVEL SUMMARY
C
263      DO 82 K=1,3
264          OUT(K)=BLANK
265- 266      82      CONTINUE
267          IF(KFT-KDT)37,37,35
268          85      IF(KTT-KDT)89,89,88
269          96      INDEX=2
270          GOTO 95
271          47      IF(KTT-KFT)89,89,86
272          88      INDEX=1
273          GOTO 95
274          89      INDEX=3
275          95      OUT(INDEX)=X
276          WRITE(IWORK2,41)IPASS,NOM,VAL(50),VAL(56),(OUT(I),I=1,3)
277          ICOUNT=ICOUNT+1
278          GOTO 200

C
C      -- COMPUTATION ROUTINE
C
279      100      TXX=QCTGM(IPASS)*VAL(16)*12.0
280          TV(1)=TXX*VAL(50)
281          PSCCON=VAL(19)+VAL(17)+VAL(21)*SQ
282          PSCOS=VAL(20)+VAL(18)+VAL(22)*SR1
283          PXX=VAL(2)*PSCCON+(1.0-VAL(2))*PSCOS
284          TV(2)=TXX*VAL(51)*PXX
285          OSTD=QCTGM(IPASS)*(VAL(2)*OSTC+UST7*(1.0-VAL(2)))
286          OSTX=(OSTD+SQRT(3.0*OSTC))
287          TV(3)=VAL(50)*OSTX

C
288          FV(1)=AIA(IPASS)+VAL(64)+VAL(63)/VAL(11)
289          FV(2)=VAL(39)*VAL(16)*(AIA(IPASS)+VAL(64))
290          FV(3)=VAL(42)*VAL(30)/VAL(11)
291          FV(4)=(1.0+VAL(34)*(VAL(16)-1.0))*(VAL(52)*VAL(55)*
-          (VAL(36)+40.0*VAL(8)))
292          A=VAL(45)*VAL(48)*12.0*QCTGM(IPASS)
293          EQQ=4.4*SQRT(A)
294          IF(EQQ-A)108,120,120
295          108      A12=A/12.0
296          IF(EQQ-A12)110,130,130
297          110      EQQ=A12
298          GOTO 130
299          120      EQQ=A
300          130      BRC=QCTGM(IPASS)*VAL(1)
301          FV(5)=VAL(50)*(BRC+SQRT(3.0*BRC))+VAL(48)*(1.0-VAL(45))*
-          OSTX+EQQ
302          FV(6)=TXX*VAL(47)*VAL(8)
303          FV(7)=TXX*VAL(49)*PXX
304          FV(8)=VAL(16)*VAL(25)*(VAL(43)+VAL(44))
305          FV(9)=VAL(40)
306          FV(10)=QCTGM(IPASS)*12.0*VAL(16)*VAL(48)
307          FV(11)=(VAL(44)-1.0)/VAL(11)*(VAL(10)+(VAL(16)-1.0)*VAL(24))+
-          VAL(43)/VAL(11)*(VAL(9)+(VAL(16)-1.0)*VAL(23))

```

C

```

308      DV(1)=QCTGM(IPASS)*VAL(5)*VAL(50)
309      DV(2)=(DAA(IPASS)+VAL(62)+VAL(61))/VAL(11)
310      DV(3)=(VAL(38)*VAL(16)*(DAA(IPASS)+VAL(62)))/VAL(11)
311      DV(4)=VAL(41)*VAL(30)/VAL(11)
312      DV(5)=(1.0+VAL(33)*(VAL(16)-1.0))*(VAL(53)*VAL(54)*
      -      (VAL(35)+40.0*VAL(4)))/VAL(11)
313      DV(6)=TXX*2.0*VAL(51)*PXX
314      DV(7)=QCTGM(IPASS)*VAL(6)*VAL(50)
315      DV(8)=TV(3)
316      DV(9)=TXX*VAL(47)*VAL(4)
317      DV(10)=VAL(37)/VAL(11)
318      DV(11)=TXX*VAL(48)
319      DV(12)=FV(11)
320      KTT=0
321      KFT=0
322      KDT=IFIX(DV(12))
323      DO 140 I=1,3
324          KTT=KTT+IFIX(TV(I))
325- 326      140      CONTINUE
327          DO 150 I=1,11
328              KFT=KFT+IFIX(FV(I))
329              KDT=KDT+IFIX(DV(I))
330- 331      150      CONTINUE
332          GOTO JUMP,(75,215,715)

      C
      C
333      200      NV=0
334          LSA=10
335          VAL(57)=DV(2)
336          VAL(58)=DV(3)
337          VAL(59)=FV(1)
338          VAL(60)=FV(2)

      C
      C
      C      -- RANK THE ECON VALUES
339      218      KSEN(1)=KDT
340          KSEN(2)=KFT
341          KSEN(3)=KTT
342          DO 335 I=1,3
343              NUMK(I)=I
344              IH=I+3
345              KSFN(IH)=KSEN(I)
346- 347      335      CONTINUE
348          DO 3100 IB=1,2
349              K=IB+1
350              DO 3100 IZ=K,3
351                  IF(KSEN(IB)-KSFN(IZ))3100,3100,304
352                  HOLD=KSEN(IB)
353                  KSEN(IB)=KSFN(IZ)
354                  KSEN(IZ)=HOLD
355                  HOLD=NUMK(IB)
356                  NUMK(IZ)=NUMK(IZ)

```

```

357                                NUMK(IZ)=HOLD
358- 360    3100    CONTINUE
              C
              C    -- SENSITIVITY ANALYSIS
              C
361    2360    NV=NV+1
362          IF(NVAR(NV))70,70,2365
363    2365    IC=NVAR(NV)
364          C=VAR(IC)
365          ORIG=VAL(IC)
366          PCT=0.90
367          DO 2300 IP=1,48
368            CX=ORIG*PCT
369            VAL(IC)=CX
370    2370    QCTGM(IPASS)=VAL(31)*VAL(32)*VAL(46)/VAL(56)
371          ASSIGN 215 TO JUMP
372          GOTO 100

              C
              C    -- REVERSAL ANALYSIS
              C
373    215     JSFN(1)=KDT
374          JSEN(2)=KPT
375          JSFN(3)=KTT
376          DO 210 I=1,3
377            NUMJ(I)=I
378- 379    210     CONTINUE
380          DO 310 IB=1,2
381            K=IB+1
382          DO 310 IZ=K,3
383            IF(JSEN(IB)-JSEN(IZ))310,310,306
384            HOLD=JSEN(IB)
385            JSEN(IB)=JSEN(IZ)
386            JSEN(IZ)=HOLD
387            HOLD=NUMJ(IB)
388            NUMJ(IB)=NUMJ(IZ)
389            NUMJ(IZ)=HOLD
390- 392    310     CONTINUE
391          IF(NUMK(1)-NUMJ(1))320,228,320
392          IF(IP-8)322,222,229
393          PCT=PCT+0.90
394          GOTO 2300
395          PCT=PCT-0.1
396          GOTO 2300
397          PCT=PCT+0.1
398          GOTO 2300
399          PCT=PCT-0.1
400          GOTO 2300
401          IF(IP-3)375,375,360
402          PCT=PCT+0.09
403          GOTO 340
404          PCT=PCT-0.19
405          CX=ORIG*PCT
406          VAL(IC)=CX
407          QCTGM(IPASS)=VAL(31)*VAL(32)*VAL(46)/VAL(56)
408          ASSIGN 715 TO JUMP

```

```

409          GOTO 100
          C
          C
          C
          715      JSFN(1)=KOT
410          JSFN(2)=KFT
411          JSFN(3)=KTT
412          DO 712 I=1,3
413          NUMJ(I)=I
414
415- 416      712      CONTINUE
417          DO 710 I9=1,2
418          K=I8+1
419          DO 710 IZ=K,3
420          IF(JSEN(I8)-JSEN(IZ))710,710,706
421          HOLD=JSEN(I8)
422          JSEN(I8)=JSEN(IZ)
423          JSFN(IZ)=HOLD
424          HOLD=NUMJ(I8)
425          NUMJ(I8)=NUMJ(IZ)
426          NUMJ(IZ)=HOLD
427- 429      710      CONTINUE
430          IF(NUMK(1)-NUMJ(1))400,350,400
431          IF(IP-9)355,355,325
432          PCT=PCT-0.01
433          GOTO 340
434          PCT=PCT+0.01
435          GOTO 340
436- 437      2300     CONTINUE
438          VAL(IC)=ORIG
439          GOTO 2360
          C
440          400      IF(LSA-10)4009,4009,4010
441          4009     WRITE(IOUTPT,49)
442          WRITE(IOUTPT,42)
443          WRITE(IOUTPT,40)
444          WRITE(IOUTPT,43)
445          4010     PCT=PCT*100.0
446          425     WRITE(IOUTPT,44)C,ORIG,(KSEN(K),K=4,6)
447          LSA=LSA+1
448          WRITE(IOUTPT,45)PCT,OX,KOT,KFT,KTT
449          VAL(IC)=ORIG
450          GOTO 2360
451          500     REWIND INWK2
          C
          C
          C
          -- WRITE REPAIR LEVEL SUMMARY
          505
452          WRITE(IOUTPT,49)
453          WRITE(IOUTPT,46)DATE(1),DATE(2)
454          WRITE(IOUTPT,47)
455          WRITE(IOUTPT,49)
456          DO 508 I=1,ICOUNT
457          READ(INWK2,41,END=510)IPASS,NOM,VAL(50),VAL(56)
          (OUT(K),K=1,3)

```

ORLA

NDAC SEGMENT XLATOR

10/31/1930 11:36 PM PAGE 16

458

WRITE(OUTPUT,411)IPASS,NOM,VAL(50),VAL(56),  
(OUT(L),L=1,3)

459- 460

508

CONTINUE

461

510

STOP

462

END

## ORLA

## Segment Reference

1. (0-13)
2. (13-20)
3. (20,16-20)
4. (20-24)
5. (24,21-24)
6. (24-55)
7. (55-60)
8. (60-61,83-122)
9. (122-130)
10. (130,86-122)
11. (130-135)
12. (135-136,200-204)
13. (204-209)
14. (209,216-228,279-294)
15. (294-296)
16. (296-298,300-326)
17. (326,323-326)
18. (326-331)
19. (331,327-331)
20. (331-332)
21. (332,229-237)
22. (237-241)
23. (241-244,246-247)
24. (247,236-237)
25. (247-266)
26. (266,263-266)
27. (266-267)
28. (267,271)
29. (271,274-278,333-347)
30. (347,342-347)
31. (347-351)
32. (351,358-359)
33. (359,350-351)
34. (359-360)
35. (360,348-351)
36. (360-362)
37. (362,204)
38. (362-372,279-294)
39. (351-359)
40. (271,269-270,275-278,333-347)
41. (267-268)
42. (268,274-278,333-347)
43. (268,272-273,275-278,333-347)
44. (241,245-247)
45. (237,241)
46. (332,373-379)
47. (379,376-379)
48. (379-383)
49. (383,390-391)
50. (391,382-393)

## RLA

## Segment Reference

- 51. [391-392)
- 52. [392,380-383)
- 53. [392-393)
- 54. [393,401)
- 55. [401-403,405-409,279-294)
- 56. [401,404-409,279-294)
- 57. [393-394)
- 58. [394,397-398,436-437)
- 59. [437,367-372,279-294)
- 60. [437-439,361-362)
- 61. [394-396,436-437)
- 62. [394,399-400,436-437)
- 63. [383-391)
- 64. [332,410-416)
- 65. [416,413-416)
- 66. [416-420)
- 67. [420,427-428)
- 68. [428,419-420)
- 69. [428-429)
- 70. [429,417-420)
- 71. [429-430)
- 72. [430,440)
- 73. [440-450,361-362)
- 74. [440,445-450,361-362)
- 75. [430-431)
- 76. [431-433,405-409,279-294)
- 77. [431,434-435,405-409,279-294)
- 78. [420-428)
- 79. [296,300-326)
- 80. [294,299-326)
- 81. [209-215,221-228,279-294)
- 82. [204,451-457)
- 83. [457-460)
- 84. [460,456-457)
- 85. [460-461]
- 86. [457,461]
- 87. [135,137-139)
- 88. [139-166)
- 89. [166,157-166)
- 90. [166-169)
- 91. [169-172,175-187)
- 92. [187-189,191-196)
- 93. [196,191-197)
- 94. [196-199)
- 95. [199,143-166)
- 96. [199-204)
- 97. [187,190-196)
- 98. [187,200-204)
- 99. [169,173-187)
- 100. [169,200-204)

## URLA

## Segment Reference

- 101. (139,141-166)
- 102. (122,124-130)
- 103. (60,62-64)
- \* 104. (64-65,62-64)
- 105. (64,66-71)
- 106. (71-72,79-81)
- 107. (81,69-71)
- 108. (81-122)
- 109. (71,73-74)
- 110. (74,70-71)
- 111. (74-78,80-81)
- 112. (55,57-60)
- 113. (13,15-20)

- NO CSSMONITOR for MODULE "URLA" -

## AGETLK

## Segment Reference

- 1. (0-21)

- NO CSSMONITOR for MODULE "AGETLK" -

## XPLAIN

## Segment Reference

- 1. (0-7)

- NO CSSMONITOR for MODULE "XPLAIN" -

## CORRECT

## Segment Reference

- 1. (0-7)
- 2. (7-8,13-16)
- 3. (16-17,0-7)
- 4. (16,18-19)
- 5. (7,9-10)
- 6. (10,6-7)
- 7. (10-12,2-7)

- NO CSSMONITOR for MODULE "CORRECT" -

## DATEV

## Segment Reference

- 1. (0-31)

- NO CSSMONITOR for MODULE "DATEV" -

A20

Appendix B  
APTS OUTPUT

	Case 1	Case 2	Case 3	Summary
MAIN	66.67 Pc	33.33 Pc	33.33 Pc	100.00 Pc
1.	1	0	0	1
2.	1	1	0	2
3.	0	0	1	1
TRIANG	86.96 Pc	86.96 Pc	86.96 Pc	36.96 Pc
1.	1	1	1	3
2.	1	1	1	3
3.	0	0	0	0
4.	2	2	2	6
5.	1	1	1	3
6.	3	3	3	9
7.	2	2	2	6
8.	1	1	1	3
9.	1	1	1	3
10.	2	2	2	6
11.	1	1	1	3
12.	0	0	0	0
13.	2	2	2	6
14.	1	1	1	3
15.	2	2	2	6
16.	0	0	0	0
17.	1	1	1	3
18.	2	2	2	6
19.	3	3	3	9
20.	2	2	2	6
21.	3	3	3	9
22.	2	2	2	6
23.	3	3	3	9
*Program	84.62 Pc	80.77 Pc	80.77 Pc	88.16 Pc

# MAIN

## Segment Reference

1. [0-11)
2. [11,5-11)
3. [11-12]

- NO CSSMONITOR for MODULE "MAIN" -

# TRIANG

## Segment Reference

1. [0-3)
2. [3-4,34)
- \* 3. [34-37)
4. [37,2-3)
5. [37-38]
6. [34,36-37)
7. [3,5-8)
8. [3-11)
9. [11,7-8)
10. [11-13)
11. [13-16)
- \* 12. [13-17,34)
13. [13,20-22)
14. [27,20-22)
15. [22-27)
- \* 16. [27-28,32-33)
17. [33,23-27)
18. [33-34)
19. [27,29-31)
20. [31,29-31)
21. [31-33)
22. [13,15-19)
23. [8,10-11)

- NO CSSMONITOR for MODULE "TRIANG" -

# Trial Statistics

Number of trials(T)= 3

Value of Xi:

Xi	11 :	1
Xi	21 :	1

Number of Xi(N)= 2

	Case	4	Case	5	Case	6	Summary
MAIN	66.67 Pc		33.33 Pc		33.33 Pc		100.00 Pc
1.	1		0		0		1
2.	1		1		0		2
3.	0		0		1		1

	Case	4	Case	5	Case	6	Summary
TRIANG	87.61 Pc		86.96 Pc		35.96 Pc		86.96 Pc
1.	1		1		1		3
2.	1		1		1		3
3.	0		0		0		0
4.	2		2		2		6
5.	1		1		1		3
6.	3		3		3		9
7.	2		2		2		6
8.	2		2		1		5
9.	1		1		1		3
10.	2		2		2		6
11.	0		1		1		2
12.	0		0		0		0
13.	2		2		2		6
14.	1		1		1		3
15.	2		2		2		6
16.	0		0		0		0
17.	1		1		1		3
18.	2		2		2		6
19.	3		3		3		9
20.	2		2		2		6
21.	3		3		3		9
22.	2		2		2		6
23.	3		3		3		9

*Program	80.77 Pc		80.77 Pc		80.77 Pc		89.46 Pc
----------	----------	--	----------	--	----------	--	----------

# MAIN

## Segment Reference

1. [0-11)
2. [11,5-11)
3. [11-12)

- NO CSSMONITOR for MODULE "MAIN" -

# TRIANG

## Segment Reference

1. [0-3)
2. [3-4,34)
- \* 3. [34-37)
4. [37,2-3)
5. [37-38)
6. [31,36-37)
7. [3,5-8)
8. [3-11)
9. [11,7-8)
10. [11-13)
11. [13-18)
- \* 12. [18-19,34)
13. [18,20-22)
14. [22,20-22)
15. [22-27)
- \* 16. [27-23,32-33)
17. [33,23-27)
18. [33-34)
19. [27,29-31)
20. [31,29-31)
21. [31-33)
22. [13,15-13)
23. [3,10-11)

- NO CSSMONITOR for MODULE "TRIANG" -

# Trial Statistics

Number of trials( $n$ )= 3

Value of  $X_i$ :

$X_i$	11 :	1
$X_i$	21 :	1
$X_i$	31 :	1

Number of  $X_i(n)$ = 3

	Case	7	Case	8	Case	9	Summary
MAIN	66.57 Pc		33.33 Pc		33.33 Pc		100.00 Pc
1.	1		0		0		1
2.	1		1		0		2
3.	0		0		1		1
TRIANG	86.96 Pc		78.26 Pc		86.96 Pc		86.96 Pc
1.	1		1		1		3
2.	1		1		1		3
3.	0		0		0		0
4.	2		2		2		6
5.	1		1		1		3
6.	3		3		3		9
7.	2		2		2		6
8.	1		0		3		4
9.	1		1		1		3
10.	2		2		2		6
11.	1		0		1		2
12.	0		0		0		0
13.	2		2		2		6
14.	1		1		1		3
15.	2		2		2		6
16.	0		0		0		0
17.	1		1		1		3
18.	2		2		2		6
19.	3		3		3		9
20.	2		2		2		6
21.	3		3		3		9
22.	2		2		2		6
23.	3		3		3		9
*Program	84.62 Pc		73.08 Pc		80.77 Pc		88.46 Pc

# MAIN

## Segment Reference

1. [0-11)
2. [11,5-11)
3. [11-12]

- NO CS\$MONITOR for MODULE "MAIN" -

# TRIANG

## Segment Reference

1. [0-3)
2. [3-4,34)
- \* 3. [34-37)
4. [37,2-3)
5. [37-33]
6. [34,35-37)
7. [3,5-3)
8. [2-11)
9. [11,7-3)
10. [11-13)
11. [13-15)
- \* 12. [18-19,34)
13. [18,20-22)
14. [22,20-22)
15. [22-27)
- \* 16. [27-28,32-33)
17. [33,23-27)
18. [33-34)
19. [27,29-31)
20. [31,29-31)
21. [31-33)
22. [13,15-19)
23. [8,10-11)

- NO CS\$MONITOR for MODULE "TRIANG" -

# Trial Statistics

Number of trials( $\tau$ )= 9

Value of  $X_i$ :

$X_i$	11 :	1
$X_i$	23 :	1
$X_i$	33 :	1
$X_i$	43 :	4

Number of  $X_i(N)$ = 4

Appendix C  
OUTPUT FROM A CONSTRUCTED CASE  
AND  
CONSTRUCTED CASES LISTINGS

# OUTPUT FROM A CONSTRUCTED CASE

	Case	1	Summary
MAIN	66.67	Pc	66.67 Pc
1.	1		1
2.	0		0
3.	1		1
TRIANG	73.91	Pc	73.91 Pc
1.	1		1
2.	1		1
3.	1		1
4.	2		2
5.	1		1
6.	3		3
7.	2		2
8.	2		2
9.	1		1
10.	2		2
11.	0		0
12.	1		1
13.	1		1
14.	0		0
15.	1		1
16.	1		1
17.	0		0
18.	1		1
19.	0		0
20.	0		0
21.	0		0
22.	2		2
23.	3		3

\*Program 73.08 Pc 73.08 Pc

# MAIN

## Segment Reference

1. [0-10)
2. [10,5-10)
3. [10-11)

- NO CDS MONITOR for MODULE MAIN -

# TRANS

## Segment Reference

1. [0-3)
2. [3-4,36)
3. [35-37)
4. [39,2-3)
5. [39-40)
6. [35,38-39)
7. [3,5-3)
8. [8-11)
9. [11,7-3)
10. [11-13)
- \* 11. [13-17)
12. [19-20,36)
13. [11,21-23)
- \* 14. [23,21-23)
15. [23-27)
16. [27-30,34-35)
- \* 17. [35,24-27)
18. [35-36)
- \* 19. [27,31-33)
- \* 20. [33,31-33)
- \* 21. [33-35)
22. [13,18-19)
23. [3,10-11)

- Monitor Predicate -

Predicate	Type	Minimum	Maximum
1. A	Integer	1	3
2. GVT	Real	.0000000000000000E+00	.4300000001000000E+01
3. FVDT	Real	.0000000000000000E+00	.0000000000000000E+00

**Trial Statistics**

**Number of trials(T)= 1**

**Value of  $X_i$ :**

**Number of  $X_i(N)= 0$**

# CONSTRUCTED CASES LISTINGS

```

PROGRAM MAIN
IMPLICIT INTEGER(A-Z)

C
INTEGER IP(3)
REAL A(4,3),R

C
DATA A/0.0,4.0,3.0,5.0,2.0,4.3,0.0,7.1,9.0,0.0,11.0,12.0

C
0- 1 OPEN(UNIT=20,DEVICE="DSK:",FILE="BRAD.RES",ACCESS="SEQOUT")
C
2 N=3
3 NN=4
4 M=1
C
5 DO 10 K=1,M
6 WRITE(20,101)K,((A(I,J),J=1,N),I=1,NN)
7 CALL TRIANG(IP,A,N)
8 WRITE(20,102)K,((A(I,J),J=1,N),I=1,NN)
9- 10 10 CONTINUE
11 STOP
C
12 101 FORMAT(' BEFORE TRIANGULARIZATION (' ,I1,')'//3(3X,F20.10))
13 102 FORMAT(' AFTER TRIANGULARIZATION (' ,I1,')'//3(3X,F20.10))
C
14 END

```

SUBROUTINE TRIANG(IP,A,N)  
 IMPLICIT INTEGER(A-Z)

C

INTEGER IP(3)  
 REAL A(4,3),T  
 REAL MONT ,TMON

C

0-	1		IP(N)=1
2			DO 6 K=1,N
			CSSMONITOR=INTEGER(K);
3-	4		IF(K.EQ.N) GOTO 5
5			KP1=K+1
6			M=K
7			DO 1 I=KP1,N
8-	9		IF(ABS(A(I,K)).GT.ABS(A(M,K))) M=1
10-	11	1	CONTINUE
12			IP(K)=M
13-	14		IF(M.NE.K) IP(N)=-IP(N)
15			T=A(M,K)
16			A(M,K)=A(K,K)
17			A(K,K)=T
18			MONT=ABS(T)
			CSSMONITOR=REAL(MONT);
19-	20		IF(I.EQ.0) GOTO 5
21			DO 2 I=KP1,N
22-	23	2	A(I,K)=-A(I,K)/T
24			DO 4 J=KP1,N
25			T=A(M,J)
26			A(M,J)=A(K,J)
27			A(K,J)=T
28			TMON=ABS(T)
			CSSMONITOR=REAL(TMON);
29-	30		IF(T.EQ.0) GOTO 4
31			DO 3 I=KP1,N
32-	33	3	A(I,J)=A(I,J)+A(I,K)*T
34-	35	4	CONTINUE
36-	37	5	IF(A(K,K).EQ.0) IP(N)=0
38-	39	5	CONTINUE
40			RETURN
41			END